Object-Centric Reflection

Unifying Reflection and Bringing It Back to Objects

Inauguraldissertation
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Jorge Ressia
von Argentina

Leiter der Arbeit:
Prof. Dr. O. Nierstrasz

Institut für Informatik und angewandte Mathematik

The logos of Bifröst and Prisma have been influenced by the design channel abduzeedo.com.

ISBN 978-1-291-07262-4
First Edition, October 2012.

Acknowledgements

First of all I would like to thank Oscar Nierstrasz for giving me the opportunity to work at the Software Composition Group. I thank him for his advice and guidance and specially for pushing me to go beyond my limitations.

I am grateful to Mira Mezini for writing the Koreferat and for accepting to be on the PhD committee, as well as for coming to Switzerland to join the jury of the PhD defense. Also, I thank Matthias Zwicker for accepting to chair the examination.

I would like to thank Gustavo Rossi and Fernando Aramburu for believing in me before I did. This thesis would have never existed without you two.

I am grateful to Stéphane Ducasse and Marcus Denker for their enthusiasm and positive support. I would like to thank them for introducing me into the reflection and meta-programming problems that this thesis addresses.

I thank Tudor Gîrba for the inspiring discussions and for providing many of the ideas that have influenced this work. I also thank Alexandre Bergel for his support, interesting dicusssions and for providing new ideas and applications for the tools presented in this thesis.

I am much obliged to the people that provided constructive feedback on early drafts of this dissertation: Orla Greevy, Fabrizio Perin, and Lukas Renggli.

This thesis would have never been completed without Fabrizio Perin and Lukas Renggli. Their emotional and technical support are directly related to the results of this thesis.

Special thanks go to Iris Keller who made the administrative work both inside and outside the university a pleasure.

I would like to thank my master students Andrea Quadri and Daniel Langone, and my bachelor students Max Leske, Manuel Leuenberger and Chantal Peeters for the hours we shared discussing and implementing Smalltalk projects and for putting up with my rather unconventional approach to research.

I would like to thank all the former and current members of the *Software Composition Group*. It was a pleasure to work with you: Andrea Caracciolo, Marcus Denker, Tudor Gîrba, Adrian Kuhn, Jan Kurš, Adrian Lienhard, Mircea Lungu, Fabrizio Perin, David Röthlisberger, Niko Schwarz, Toon Verwaest, and Erwann Wernli.

I thank the people with whom I shared many interesting discussions at conferences: Nick Ager, Marco D'Ambros, Michele Lanza, Philippe Marschall, Fernando

Oliveros, Serge Stinckwich and Adrian van Os. Special thanks to the early fanatical Bifröst supporters and believers who kept me going: Alexandre Bergel, Noury Bouraqadi, Jordi Delgado, Simon Dennier, Luc Fabresse, Dale Henrichs, Rafael Luque and Erwann Wernli.

I would like to deeply thank Angela for making us feel like home. Many, many thanks to my family. Thanks for your love and support: Ricardo, Cristina, Mariana, Luciana, Isabella, Roman, Hernan and Adrian. Above all, I thank my wife Analia Magurno for putting up with me, which is quite difficult.

Jorge Ressia
October 8, 2012

To Carmela and Nelly

Abstract

Reflective applications are able to query and manipulate the structure and behavior of a running system. This is essential for highly dynamic software that needs to interact with objects whose structure and behavior are not known when the application is written. Software analysis tools, like debuggers, are a typical example. Oddly, although reflection essentially concerns run-time entities, reflective applications tend to focus on static abstractions, like classes and methods, rather than objects. This is phenomenon we call the *object paradox*, which makes developers less effective by drawing their attention away from run-time objects.

To counteract this phenomenon, we propose a purely *object-centric* approach to reflection. Reflective mechanisms provide object-specific capabilities as another feature. Object-centric reflection proposes to turn this around and put object-specific capabilities as the central reflection mechanism. This change in the reflection architecture allows a unification of various reflection mechanisms and a solution to the object paradox.

We introduce Bifröst, an object-centric reflective system based on first-class meta-objects. Through a series of practical examples we demonstrate how object-centric reflection mitigates the object paradox by avoiding the need to reflect on static abstractions. We survey existing approaches to reflection to establish key requirements in the domain, and we show that an object-centric approach simplifies the meta-level and allows a unification of the reflection field. We demonstrate how development itself is enhanced with this new approach: *talents* are dynamically composable units of reuse, and *object-centric debugging* prevents the object paradox when debugging. We also demonstrate how software analysis is benefited by object-centric reflection with Chameleon, a framework for building object-centric analysis tools and MetaSpy, a domain-specific profiler.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A reflective computational system is capable of inspecting, manipulating and altering its representation of itself [Smith, 1982]. Reflection is commonly used to implement development tools such as debuggers and profilers, and to realize run-time adaptations for highly dynamic applications that, for example, must generate user interfaces at run-time. A reflective system can be divided into two levels: the base level, which is concerned with the application domain, and the meta-level, which encompasses the self-representation. These levels are causally connected, so any modification to one level affects any further computation on the other level. There are two types of reflection: structural reflection is concerned with the manipulation of structural elements of a program while behavioral reflection is concerned with the manipulation of the abstractions which govern the execution of a program. These structural and behavioral abstractions can be queried (introspection) and changed (intercession) from within the running system.

Reflective applications are able to query and manipulate the structure and behavior of a running system. This is essential for highly dynamic applications that need to interact with objects whose structure and behavior are unknown when the application is written.

1.1 Reflection Requirements

In recent years researchers have worked on the idea of applying traditional engineering techniques to the meta-level while attempting to solve various practical problems motivated by applications [McAffer, 1996]. These approaches, however, offer specialized solutions arising from the perspective of particular use cases. We have analyzed these approaches and identified six distinct and key requirements for a meta-level architecture. These requirements are supported only partially by existing approaches:

1. *Partial Reflection* makes reflective facilities available only in selected places where needed. This avoids the inherent inefficiency of a fully reflective system [Ibrahim, 1991; Kiczales *et al.*, 1991; Tanter *et al.*, 2003].

2. *Selective Reification* refers to the ability to define which reifications should be active from a temporal and spatial point of view. Selective reification extends partial reflection to allow reifications to be dynamically defined [Ferber, 1989; Gowing and Cahill, 1996; Redmond and Cahill, 2002; Redmond and Cahill, 2000].

3. *Unanticipated Changes* enable reflection on a running system without the need to define statically and up-front where and when reflection is needed [Redmond and Cahill, 2002; Redmond and Cahill, 2000; Denker, 2008; Denker *et al.*, 2007].

4. *Runtime Integration* refers to a meta-environment that runs at the same level as the application code, *i.e.*, not in the interpreter of the host language [Tanter *et al.*, 2003; Denker, 2008; Bouraqadi, 2004].

5. *Meta-level Composition* enables the combination of meta-level abstractions due to multiple adaptations taking place on the same base-level abstractions [Tanter, 2006; Bobrow *et al.*, 1988; Kiczales *et al.*, 1991; Redmond and Cahill, 2002; Redmond and Cahill, 2000].

6. *Scoped Reflection* makes reflective changes only visible in specific contexts; outside these contexts the changes are not present [Chiba *et al.*, 1996; Aracic *et al.*, 2006; Denker *et al.*, 2008; Tanter, 2009].

No current approach supports all of these requirements. This is problematic because certain problems can be solved by some approaches and not by others.

1.2 The Problem

Object-oriented languages and methods encourage the design of software systems in terms of interacting and collaborating objects. Developers of object-oriented applications, however, spend most of their time interacting not with objects, but with purely static abstractions, namely classes and methods in the form of source code. Integrated development environments and related tools tend to focus on the static source code rather than on the running system. We can also observe this problem in the area of reflection. Development tools, like debuggers and profilers, are classical tools that must use some form of reflection to interact with arbitrary applications. Although the goal of reflection is to enable run-time adaptation, reflective mechanisms tend to focus on representation of static artifacts, *i.e.*, related to the source code, rather than on the run-time entities, *i.e.*, the objects. When we look deeper into how languages implement reflective applications we observe a chronic pattern to move away from the runtime abstractions towards static ones.

This is a problem since the developer needs to express his needs in terms of the object's static representation instead of directly reflecting on the object. There is an

unnecessary indirection through objects' static representations to reflect on these objects. Next we will analyze three examples to clarify this statment.

Debugging

Debugging is formally the process of finding and reducing the number of defects in a computer program, thus making it behave as expected. More broadly, however, debugging is the process of interacting with a running software system to test and understand its current behavior. Software developers frequently turn to debuggers to obtain insight into parts of a running system before attempting to change it, rather than to remove defects. Similarly, in test-driven development [Beck, 2002], debuggers are frequently used as a development tool to identify those parts of the system that need to be implemented next.

Traditional debuggers are focused on the *execution stack*. The developer identifies parts of the source code of interest and sets *breakpoints* accordingly. The software then runs until a breakpoint is reached, and the developer can then inspect and interact with the code and entities in the scope of the breakpoint. Unfortunately this process is ill-matched to typical development tasks. Breakpoints are set purely with respect to static abstractions, rather than to specific objects of the running system. As a consequence, identifying the right place to set breakpoints in the source code requires a deep understanding of what happens during the execution. Second, debugging operations are focused on the execution stack, rather than on the objects. There exists therefore a considerable conceptual gap between the interface offered by the debugger and the questions of interest by the developer. In the debugger we deal with objects constantly, but when we need to execute a debugging action we jump out of the runtime environment into the static abstractions.

By forcing developers to work with static abstractions, they become less effective in debugging.

Profiling

Current application profilers are used to gather runtime data (*e.g.*, method invocations, method coverage, call trees, code coverage, memory consumption) from the static code model offered by the programming language (*e.g.*, packages, classes, methods, statements). This is an effective approach when the low-level source code has to be profiled.

However, traditional profilers are far less useful for a domain different than the code model. In modern software there is a significant gap between the model offered by the execution platform and the model of the actually running application.

The proliferation of meta-models and domain-specific languages brings new abstractions that map to the underlying execution platform in non-trivial ways. Traditional profiling tools fail to display relevant information in the presence of such abstractions.

Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the current set of methods under execution. Such a profiling technique is relatively accurate since it has little impact on the overall execution. This sampling technique is used by almost all mainstream profilers, such as JProfiler, YourKit, xprof [Gupta and Hwu, 1992], and hprof.

Traditional execution sampling profilers center their result on the frames of the execution stack and completely ignore the identity of the object that is the target of the method call and its arguments. As a consequence, it is hard to track down which objects cause the performance slowdown that triggered the profile. For the example above, the traditional profiler states how much time was used by a particular method in a class without saying which objects were actually involved.

Traditional profilers provide static-related information that is suboptimal for finding the time consumed by each object.

Feature Analysis

A feature represents a functional requirement fulfilled by a system. Since many maintenance tasks are expressed in terms of features, it is important to establish the correspondence between a feature and its implementation in source code.

Many researchers have recognized the importance of centering reverse engineering activities around a system's behavior, in particular, around features [Eisenbarth *et al.*, 2003; Kothari *et al.*, 2006; Salah and Mancoridis, 2004]. Bugs and change requests are usually expressed in terms of a system's features, thus knowledge of a system's features is particularly useful for maintenance [Mehta and Heineman, 2002].

Features are abstract notions, normally not explicitly represented in source code or elsewhere in the system. Therefore, to leverage feature information, we need to perform feature analysis to establish which portions of source code implements a particular feature. Most existing feature analysis approaches [Salah and Mancoridis, 2004; Kothari *et al.*, 2006] capture traces of method events that occur while exercising a feature and subsequently perform post-mortem analysis on the resulting feature traces.

A post-mortem feature analysis implies a level of indirection from a running system. This makes it more difficult to correlate features and the relevant parts of a running system. We lose the advantage of interactive, immediate feedback which we would obtain by directly observing the effects of exercising a feature. Post-mortem analysis does not exploit the implicit knowledge of a user performing acceptance testing. Certain subtleties are often only communicated to the system developer when

the user experiences how the system works while exercising the features. These approaches typically generate large amounts of data to analyze. Due to their static nature, these approaches do not support incremental and interactive analysis of features. Clearly, in this case, a model-at-runtime of features, with the added ability to "grow" the feature representation as the user exercises variants of the same feature offers advantages of context and comprehension over a one-off capture of a feature representation and post-mortem analysis.

1.3 Problem Statement

These three examples offer a glimpse of the general sate when developing reflective applications. In the case of feature analysis the approach is defined as a purely static problem, since we need to detect which source entities were executed. However, as we showed in live-feature analysis [Denker *et al.*, 2010] approach, having a purely dynamic approach to feature analysis delivers important advantages. Profiling is a dynamic problem which is generally solved statically. Domain-Specific profiling [Ressia *et al.*, 2012b] showed that profiling and providing information about dynamic abstraction is more meaningful to the developer. The debugging case is paradigmatic in the sense that it clearly shows an unnecessary jump out of the runtime environment. When debugging, many developer questions are targeted to the live objects not to their static representation. These examples show different levels of static approaches to dynamic reflection problems, however, to solve this problem we required a brand new approach to reflection.

Researchers have detected a similar problem in the realm of IDEs. Due to the narrow focus of IDEs on static source perspectives, most of dynamic relationships between source artifacts remain unclear, obscure or simply invisible to the developer while using the static perspectives of IDEs [Röthlisberger, 2010]. In short, traditional IDEs lack dynamic information in their usually purely static source perspectives. Object-oriented language features such as late-binding, inheritance, or polymorphism, usually lead to distributed and scattered code which is hard to understand by just focusing on static source artifacts and static relationships between these artifacts [Demeyer *et al.*, 2003; Dunsmore *et al.*, 2000; Wilde and Huitt, 1992; Nielson, 1989; Hamou-Lhadj *et al.*, 2005]. Often it is not possible to identify and locate conceptually related code in the static source space as many relationships are purely dynamic and thus only present at runtime [Nielson, 1989; Nielsen and Richards, 1989; Dunsmore *et al.*, 2000].

Providing dynamic information to enhance the IDEs' purely static source perspective is important. However, for some problems like feature analysis, debugging and profiling it is not enough with dynamic information. Why do we analyze the runtime abstractions from a dynamically enhanced source code perspective when we can directly deal with runtime objects? IDEs make heavy use of reflection to achieve

their goals. Most programming languages present reflective mechanisms for inspecting and modifying the internals of the language itself. Even though some of these mechanisms are highly flexible and capable of changing the behavior of single objects, reflective applications built on top of them fail to embrace the full dynamicity that is required in some cases. The main problem is not what extra dynamic information the IDE provides to the user, the main problem is that if we want to analyze the runtime we must do it directly looking at objects. IDEs rely on reflective mechanisms which are targeted to the wrong abstractions.

We call this problem the *object paradox*: although object-oriented developers are supposed to think in terms of objects, the tools and environments they use mostly prevent this. As we have seen in the previous examples runtime objects are not the first options from a tool perspective. The object paradox makes us less effective as developers. A developer needs to understand the run-time behavior of interacting objects in order to reason about the effects of changes to the system, but the IDE presents only static abstractions, such as classes and their specialization hierarchies, or methods and source code. This gap forces the user to adapt the system with *ad hoc* methods, like conditional breakpoints in debugging, for avoiding the paradox. Thus rendering the user less efficient than he could actually be.

We can also observe the object paradox in the area of reflection. Reflection is needed wherever an application must deal at run time with objects that are unknown to it at compile time. A reflective application, for example, may dynamically generate a graphical user interface for an object whose structure and behavior is loaded at run time. Development tools, like debuggers and profilers, are classical tools that must use some form of reflection to interact with arbitrary applications. Although the goal of reflection is to enable run-time adaptation, reflective mechanisms tend to focus on representation of static artifacts, *i.e.*, related to the source code, rather than on the run-time entities, *i.e.*, the objects. Reflection mechanisms are not object-centric per se forcing the developer to move away from the runtime. Moreover reflection mechanisms do not provide a unified approach thus forcing the user to deal with several different techniques to introspect and intersect and application.

Reflective systems prioritizing static mechanisms over object reflection present a gap between the user needs and what the reflective systems provides. Thus, the user is less efficient since he has to introduce *ad hoc* changes to steer the reflective systems to solve its object-specific needs.

1.4 Challenges

The challenges that we face in the reflection domain are:

Reflection Targeted Abstractions. To close the gap between the developer's needs and the reflection mechanism objects must be the central target of reflection changes. Since the static representation like classes and source code are also

objects in the system, traditional reflection on these abstractions is also achievable.

Unified Reflection Approach. A fully general approach to reflection must support the reflection requirements: partial reflection, selective reifications, unanticipated changes, runtime integration, meta-level composition and scoped reflection.

Uniform Reflection Approach. The mechanism for adapting objects should be consistent. It is extremely undesirable to have various reflection mechanisms depending on the objects being reflected on.

1.5 Thesis Statement

We state our thesis as follows:

> **Thesis**
> *To overcome the object paradox while providing a unified and uniform solution to the key reflection requirements we need an object-centric reflective system which targets specific objects as the central reflection mechanism through explicit meta-objects.*

1.6 Our Solution in a Nutshell

This dissertation tackles the object paradox in the reflection domain. We present Bifröst, an object-centric reflective system that offers fine-grained unanticipated dynamic structural and behavioral reflection based on explicit meta-objects. Reflective changes are object-centric, meta-objects are tailored to specific objects. Explicit meta-objects allow us to provide a range of reflective features and thereby evolve both application models and the host language at run-time. Furthermore, by simplifying the meta-level, Bifröst offers a unified approach to reflection.

Organizing the metal-level behavior into meta-objects has been extensibly researched and it is area known as Meta-object architecture [Maes, 1987b; Maes, 1987a]. What this thesis proposes as *new* is the purely reflective object-centric Bifröst approach. We are not claiming that by only having a mechanism for applying reflective changes on specific objects we can solve the previously presented problems. Several reflective techniques like MOPs CLOS, Ruby, Smalltalk anonymous classes and dynamic aspect are already capable of doing that (refer to Chapter 2 for further details). We claim that reflective changes should only be targeted to objects, they should be object-centric, then more complex meta-level abstractions like classes, prototypes, mixins, traits, can be built upon this. Bifröst only allows the user to change single objects' structure and behavior by making objects the central actors. Thus, we are capable of perceiving known problems like feature analysis, profiling, debugging

and scoped reflection from a completely different point of view. Previous solutions thought to be dynamic are shown to be partially static. This new meta-object adaptation enhances the capacity of the user on top of his objects' domain.

The Bifröst model provides an object-centric approach while supporting the main reflection requirements.

- *Partial Reflection.* Bifröst allows meta-objects to be bound to any object in the system thus reflecting selected parts of an application.
- *Selective Reification.* When and where a particular reification should be reified is managed by the various meta-objects.
- *Unanticipated Changes.* At any point in time a meta-object may be bound to any object thus supporting unanticipated changes.
- *Runtime Integration.* Bifröst reflective model lives entirely in the language model, so there is no VM modification or low level adaptation required.
- *Meta-level Composition.* Composable meta-objects provide the means to bring together different adaptations.
- *Scoped Reflection.* Meta-objects reflective changes can be scoped to particular dynamic extents and conditions.

Figure 1.1 depicts the layered architecture of the Bifröst and displays the chapters in which the respective parts are discussed.

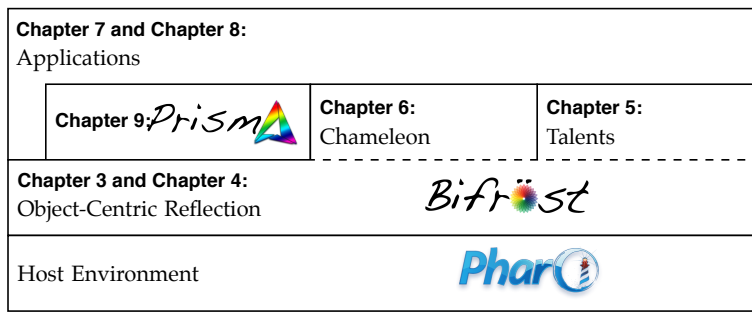

Figure 1.1: The architecture of Bifröst reflective system.

Host Environment. At the lowest layer we have the host language and its tools. In our case, this is Pharo Smalltalk [Black *et al.*, 2009], a dynamically typed object-oriented programming language with an integrated development environment. While Smalltalk [Goldberg and Robson, 1989] has proven to be a good practical choice for Bifröst it is not a requirement.

Bifröst. The layer above the host environment is the core of the Bifröst system. This layer provides the necessary hooks into the host language compiler and the tools supplied with the development environment. Bifröst realizes a simple meta-object architecture where reflective changes are object-centric. Meta-objects can be applied to single specific objects, unanticipatedly, selectively reifying runtime abstractions and composed to form more complex meta-level abstractions and adaptations.

Talents. Talents are object-specific units of reuse that model features that an object can acquire at run-time. Like a trait [Schärli *et al.*, 2003; Ducasse *et al.*, 2006b], a talent represents a set of methods that constitute part of the behavior of an object. Unlike traits, talents can be acquired (or lost) dynamically. When a talent is applied to an object, no other instance of the object's class is affected. Talents may be composed of other talents, and, as with traits, the composition order is irrelevant. Conflicts must be explicitly resolved. Talents are built on top of Bifröst's structural meta-objects. Talents address the object-centric structural reflection domain, in particular, they address all the reflection requirements but scoped reflection.

Chameleon. Chameleon provides a full operational decomposition [McAffer, 1996] of the meta-level, separating instrumentation from analysis with the help of explicit meta-level events. The meta-level's behavioral model is simplified by offering a single canonical event which models the execution of an abstract syntax tree (AST) node. Any other object-related event can be expressed in terms of this canonical event. Objects in an application are instrumented to reify meta-level events. Analysis tools select which events to observe for the purpose of profiling, logging, coverage, *etc.* Chameleon is built on top of Bifröst's behavioral meta-objects. Chameleon addresses object-centric behavioral reflection, it supports all the reflection requirements but scoped reflection.

Prisma. Prisma, an approach to support *dynamic, scoped, and live reflection on running systems*. By using and extending Bifröst meta-objects Prisma addresses the scoped reflection requirement. The central idea of Prisma is to dynamically install reflective meta-objects on the objects reached by a running software system to adapt their behavior. Prisma's meta-objects are scoped to individual objects and threads, though their scope can be enlarged to whole classes or other threads if needed. The dynamic scope is reified thus allowing the user to reflect upon and adapt the scope itself. Multiple adaptations can be simultaneously installed to enable multiple non-interfering analyses. Meta-objects are responsible for deciding which should be the behavior and structure of an object under a specific dynamic scope. Installation is decoupled from deinstallation, so adaptations can be retained to support long-lived, iterative and incremental analyses.

Tools. On the top layer reflective applications can be defined taking advantage of Bifröst object-centric reflection approach. For example MetaSpy [Ressia *et al.*,

2012b] is a domain-specific profiler which closes the gap between the domain and the profiler information. Object-centric debugging [Ressia *et al.*, 2012a] provides the developer with object-centric actions for dealing directly with runtime objects instead of having to translate the developer needs to the static domain with conditional breakpoint or similar constructs. We developed all these tools on top of Bifröst's object-centric reflection approach. Debugging and profiling are canonical examples of applications of reflection. Debugging is directly related to the development scope, moreover, it is an interesting example since runtime execution, development and live interaction come together. Profiling is a typical examples of dynamic application analysis.

1.7 Contributions

The main contributions of this dissertation are:

1. We present *Bifröst,* an object-centric reflection approach which overcomes the object paradox. Bifröst models meta-objects explicitly, exclusively targeting objects as the sole reflective change unit. This model provides a unification of different reflection approaches while solving the most important reflection requirements: partial reflection, selective reifications, unanticipated changes, runtime integration, meta-level composition and scoped reflection [Ressia *et al.*, 2010].

2. We propose *Talents*, a new approach that deals with reuse at the object level and that supports behavioral and state composition. We introduce a new abstraction called a *talent* which models behavior and state that are shared between objects of different class hierarchies. Talents provide a composition mechanism that is as flexible as that of traits but which is dynamic [Ressia *et al.*, 2011].

3. We demonstrate *Chameleon*, a tool modeling the meta-level as explicit meta-events observable by development tools. Chameleon provides an operational decomposition of the meta-level. Instrumentation is dedicated to generating meta-events, and is fully separated from analysis tools which selectively subscribe to these events by applying the observer pattern at the meta-level.

4. We present *Prisma*, an approach to support *dynamic, scoped, and live reflection on running systems*. Prisma dynamically installs reflective meta-objects on the objects reached by a running software system to adapt their behavior. Prisma's meta-objects are scoped to individual objects and threads, though their scope can be enlarged to whole classes or other threads if needed. The dynamic scope is reified thus allowing the user to reflect upon and adapt the scope itself.

The following list details the contributions with some extended case studies, which serve as the validation of our approach:

Domain-specific profiling. We presented MetaSpy, a framework for defining domain-specific profilers. We also presented three real-world case-studies showing how MetaSpy fulfills the domain-specific profiler requirements. The use of Bifröst makes it possible to instrument specific objects to provide runtime abstractions related to profiling information [Ressia *et al.*, 2012b].

Object-centric debugging. We close the gap between developers' questions and the debugging tool by shifting the focus in the debugger from the execution stack to individual objects. The essence of object-centric debugging is to let the user perform operations directly on the objects involved in a computation, instead of performing operations on the execution stack. Bifröst's meta-object were used to apply object-specific breakpoints dynamically to drive the debugger from within the runtime environment [Ressia *et al.*, 2012a].

Scoped back-in-time debugger. This technique allows developers to step both forward and backward through an entire execution run. We show how the *object-flow analysis* approach to back-in-time debugging, previously supported by VM modifications, is easily implemented using Prisma's scoped meta-objects.

Scoped live-feature analysis. Software artifacts that implement a given feature are identified by instrumenting the system and exercising those features. By using Prisma we avoid the need to statically instrument the entire system. Furthermore, multiple features can be exercised at the same time, since Prisma scopes the effect of adaptations to individual execution runs.

1.8 Outline

The dissertation is structured as follows:

Chapter 2 discusses the related work of this thesis. We present various approaches to reflection and analyze their characteristics.

Chapter 3 presents the object-centric reflection model and explains how explicit meta-objects can be used to provide object-centric reflection.

Chapter 4 introduces the Bifröst object-centric reflection implementation and validates this model through a series of examples.

Chapter 5 presents dynamically composable units of reuse called talents.

Chapter 6 introduces an operational decompostion of the meta-level called Chameleon.

Chapter 7 presents MetaSpy, a domain-specific profiler which brings profiling results closer to the domain being analyzed.

Chapter 8 introduces a new debugging technique called object-centric debugging. Developers do not have to leave the runtime environment when debugging by using object specific actions.

Chapter 9 demonstrate how Bifröst meta-objects reflective changes can be scoped to dynamic extents with Prisma.

Chapter 10 outlines our conclusions and identifies future work.

Appendix A describes how to get started with Bifröst and the related tools.

Figure 1.2 displays schematically the chapters and the problem space they cover.

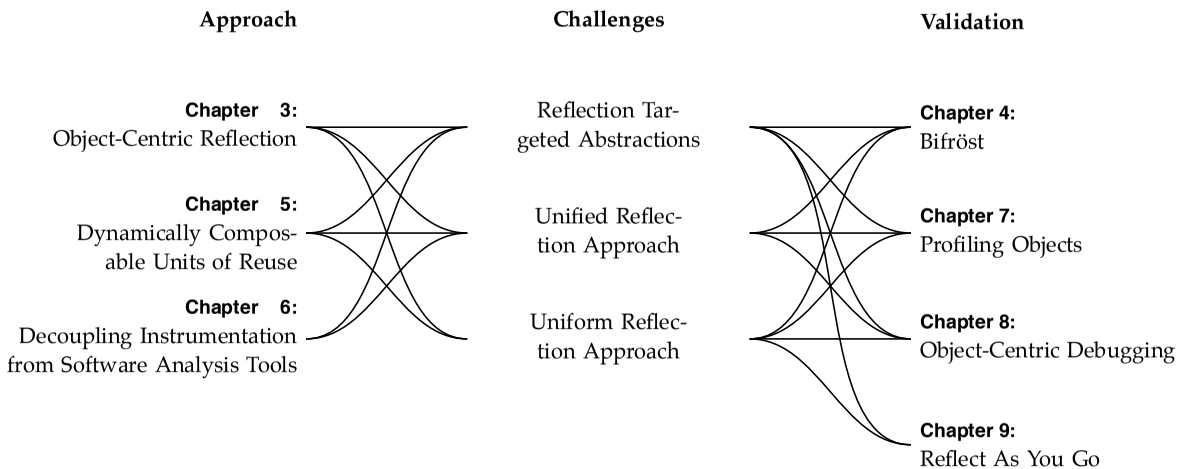

Figure 1.2: Structure of the dissertation and how it covers the problem space.

Chapter 2

Reflection State Of The Art

In this section we survey the evolution of reflective facilities in various programming languages. We present the applications of reflection. We summarize the practical problems each new reflection approach has been designed to tackle. We demonstrate that no approach solves all reflection requirements. Finally, we show that object-specific reflection is seen as a particular case of the reflection problem instead of being the central reflection mechanism. This is one of the main reasons why the object paradox is present in the reflection domain.

2.1 Applications of Reflection

Nowadays object-oriented languages and environment use heavily reflection. There are two main groups of reflective applications in these languages: *program analysis* and *development*.

Program Analysis. These applications use reflection for querying a system either from a static or dynamic point of view. Examples of these applications are: code coverage, profiling, feature analysis, metrics, *etc.*

Development. This group of applications use reflection to enhance or modify the way developments is being done. For example source code browsers and editors help the developer to have an enhanced view on the application. On the other hand debugging, code generation, dynamic testing, mock generation, parallelization, database mappings, *etc.*, allow the user to interact with the system from a dynamic point of view. Finally, language extensions like traits or mixins are reflection applications that allow the language to evolve.

In this dissertation we use canonical reflection applications to demonstrate our points of view. From program analyses we use feature analysis and profiling two traditional and highly used examples in the domain. From development we use debugging, a very special reflection application since it is one of the most used development tools and mixes the static point of view with the dynamic execution of the application.

2.2 Reflection Dimensions

Numerous approaches to reflection have been developed over the years, each of which addresses a different domain of reflection. In this section we analyze the different dimensions that can be used to categorize a reflective system.

2.2.1 Definition

A reflective system is a system which incorporates causally connected structures representing (aspects of) itself [Maes, 1987b]. A system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, this leads to a corresponding effect of the other. A reflective language thus has a representation of its own structure and behavior available from within. The representation changes if the language changes and vice versa. It is always in sync with the system itself. Therefore, the representation can be queried and it can even be changed.

Many programming languages provide mechanisms to query a representation of the system, known as introspection. Intercession is the mechanisms that allows a programming language to change the representation of itself. Only when we can both query and change the representation, we call the system reflective.

2.2.2 Elements

The literature splits reflection into two large categories [Ferber, 1989]: *structural reflection* is concerned with the manipulation of structural elements of a program while *behavioral reflection* is concerned with the manipulation of the abstractions which govern the execution of a program. In an object-oriented language adding a method or adding an instance variable to a class is an example of structural reflection. Behavioral reflection could for instance give access to base-level operations such as method calls, instance variables accesses, as well as the state of execution.

Behavioral and structural reflection can be seen on the one hand as orthogonal concepts: a language can provide functionality for behavioral or structural reflection or both. On the other hand, they are connected: any change of structure leads to a change of behavior and any behavioral change needs to change structure at some level.

As a structural change can be used to change behavior, structural reflection can serve as the basis for behavioral reflection. One example for this is MethodWrappers [Brant *et al.*, 1998], which allows methods to be wrapped to execute additional code before or after the method. Another example is Reflex [Tanter *et al.*, 2003] which realizes behavioral reflection by transforming bytecode.

2.2.3 Models

Two core models of object-oriented structural reflection have been proposed in the past, one based on *meta-classes*, *i.e.*, classes whose instances are classes, and the other on *meta-objects*, *i.e.*, objects that describe or manipulate other objects. Languages based on these models traditionally provide support for reflecting on a fixed set of language constructs. A third model diverges from the previous two because it reifies the action of sending a message, thus it is closer to behavioral reflection.

Meta-class Model. In this model the class of an object is considered to be its meta-object, since it is responsible for defining its structure and behavior. Every class is an instance of a meta-class. Since meta-classes specify the structure and behavior of classes, they are the meta-objects of classes. Some variants of this model enforce all classes to be instances of a unique meta-class, as in Smalltalk-76 [Ingalls, 1978] and Java. In other systems, like Smalltalk-80 [Goldberg and Robson, 1989], ObjVLisp [Cointe, 1987] and Classtalk [Briot and Cointe, 1989], each class is a unique instance of its meta-class. The main drawback of the meta-class model is that per-object specialization is not possible. Any change to a class impacts all instances of that class. It is not possible to go to a more fine-grained level than a class, *i.e.*, methods and operations. Composition is not possible since no object can have multiple classes. Each class share the same message interpreter: there is no possibility to specialize the interpreter for a unique object. Metaclass substitution is dangerous and can quickly lead to inconsistencies. Finally, a class cannot keep specific characteristics of specific objects.

Meta-object Model. In this model every object has its own unique meta-object. This model was first proposed by Maes in 3-KRS [Maes, 1987b; Maes, 1987a]. Since it was conceived for a prototype-based language, the notion of classes was not supported. Ferber [Ferber, 1989] analyzed how a meta-object model would behave with the introduction of classes. Behavioral and structural reflection are separated, and classes handle the definition of the structure and the set of messages that an instance is able to answer. Meta-objects handle how messages are interpreted. This model is more flexible than the meta-class model. By modifying the meta-object we can achieve per-object specific behavior, object monitoring, and different message interpretation techniques. However, this approach is mainly concerned with modeling structural constructs, neglecting the behavioral abstraction. For example, method calls and instance variables accesses are not reified.

Message Reification. This model reifies the messages sent between objects. Ferber [Ferber, 1989] introduced this model where each message is an instance of a message class. Each message is responsible for interpreting itself. The message class defines a message send specifying the interpretation. Through the message class sub-classification the message send semantics can be modified. In Ferber's model the sender of the message was not taken into account in

the reification. Cazzola [Cazzola, 1998] extended this model by including the sender object in the message reification.

2.2.4 Constructs

There are two main approaches to specifying which constructs may be reflected upon. There are *interpreter-based* approaches like 3-Lisp [Smith, 1982] and 3-KRS where meta-objects match the structure of the interpreter; and *language-oriented* approaches like CLOS-MOP [Bobrow *et al.*, 1988; Kiczales *et al.*, 1991], ObjVLisp and Classtalk, where the meta-objects match structural elements of the language. This *structural* point of view contrasts with the *computational* or *behavioral* point of view.

Smith [Smith, 1982; Smith, 1984] pioneered the concept of behavioral reflection in the context of Lisp. He proposed reifications, such as method invocations, that were not directly reflected in the structure of the language. Of course, both interpreter-based and language-based approaches can achieve behavioral reflection but there is no generalized infrastructure for doing this [Ferber, 1989; McAffer, 1995a].

2.3 State of the Art in Meta-level Engineering

Table 2.1 summarizes previous meta-level engineering approaches. In this table we show to which extent previous approaches support the four key application requirements of *partial reflection*, *selective reification*, support for *unanticipated changes* and offering a *runtime integration*. We also identify how the various approaches fall short in supporting the meta-level engineering requirements of offering an *unbiased reflective model*, providing *high-level abstractions*, and offering a means for *meta-level composition*.

2.3.1 Reflection requirements

The reflective requirements that we have pointed out are not new. In recent years researchers have worked on the idea of applying traditional engineering techniques to the meta-level while attempting to solve various practical problems motivated by applications. We will contrast object-specific reflection and object-centric reflection to stress the reasons for the existence of the object paradox. These requirements are supported at least partially by existing approaches:

1. *Partial Reflection* makes reflective facilities available only in selected places where needed. This avoids the inherent inefficiency of a fully reflective system [Ibrahim, 1991; Kiczales *et al.*, 1991; Tanter *et al.*, 2003].

Type	Reflective System	Host Language	Partial Reflection	Selective Reification	Unanticipated Changes	Runtime integration	Meta-level Composition	Scoped Reflection	Object-specific Reflection
Language Extensions	ClassTalk	Smalltalk	●	●	●	●	●	○	○
	CodA	Smalltalk	○	○	○	●	○	○	○
	Dynamic AOP	Various	●	●	●	○	●	○	●
	Guarana	Java	●	●	○	●	○	○	○
	Iguana	C++	●	●	○	○	●	○	●
	Iguana/J	Java	●	●	●	○	●	○	●
	Kawa	Java	●	●	○	●	○	○	○
	MetaClassTalk	Smalltalk	●	○	●	●	●	○	○
	MetaXa	Java	●	●	○	●	○	○	○
	PBI	Java	●	●	●	○	○	●	●
	Reflective Java	Java	●	●	○	●	●	○	○
	Reflex	Java	●	●	○	●	○	●	○
	Reflectivity	Smalltalk	●	●	●	●	○	●	○
Language Implementations	3-Lisp	3-Lisp	○	○	●	○	●	○	○
	3-KRS	3-KRS	○	○	●	○	●	○	○
	CLOS	CLOS	●	●	●	○	●	○	●
	Cola	Cola	●	●	●	○	●	○	●
	Java	Java	○	○	○	○	●	○	○
	ObjVLisp	ObjVLisp	●	○	○	○	●	○	○
	Ruby	Ruby	●	●	●	○	●	○	●
	Smalltalk-80	Smalltalk	○	○	●	○	●	○	○
	Self	Self	●	●	●	○	●	○	●

Table 2.1: Comparison of different language and reflection extensions.

2. *Selective Reification* refers to the ability to define which reifications should be active from a temporal and spatial point of view. Selective reification extends partial reflection to allow reifications to be dynamically defined [Ferber, 1989; Gowing and Cahill, 1996; Redmond and Cahill, 2002; Redmond and Cahill, 2000].

3. *Unanticipated Changes* enable reflection on a running system without the need to define statically and up-front where and when reflection is needed [Redmond and Cahill, 2002; Redmond and Cahill, 2000; Denker, 2008; Denker *et al.*, 2007].

4. *Runtime Integration* refers to a meta-environment that runs at the same level as the application code, *i.e.*, not in the interpreter of the host language [Tanter *et al.*, 2003; Denker, 2008; Bouraqadi, 2004].

5. *Meta-level Composition* enables the combination of meta-level abstractions due to multiple adaptations taking place on the same base-level abstractions [Tanter, 2006; Bobrow *et al.*, 1988; Kiczales *et al.*, 1991; Redmond and Cahill, 2002; Redmond and Cahill, 2000].

6. *Scoped Reflection* makes reflective changes only visible in specific contexts, outside these contexts the changes are not present [Chiba *et al.*, 1996; Aracic *et al.*, 2006; Denker *et al.*, 2008; Tanter, 2009].

Partial Reflection

Full reflection, where all constructs that may be reflected upon are reified, is inherently inefficient. *Partial reflection* was first introduced in the 1990 OOPSLA / ECOOP workshop on Reflection and Meta-level Architectures in Object-Oriented Programming [Ibrahim, 1991]. *Partial reflection* overcomes this inefficiency by making reflective facilities available only where they are needed. For example, we can reify the method lookup for a single class and not for all classes in the system.

Kiczales *et al.* [Kiczales *et al.*, 1991] introduced meta-object protocols (MOPs) in CLOS, an object-oriented extension of Lisp. MOPs encode the properties and semantics of the language. The MOP is causally connected to the language model. MOPs provide a form of partial reflection since they offer a means to adapt the meta-level behavior for selected parts of the system. Partial reflection can be achieved by specializing the meta-class generic functions for a specific meta-object class [Attardi *et al.*, 1989]. However, CLOS-MOP does not support object-specific method invocation reification in a scalable way, as McAffer [McAffer, 1995a] pointed out.

Partial Behavioral Reflection was introduced by Tanter *et al.* [Tanter *et al.*, 2003]. This model is implemented in Reflex for the Java environment. Reflex offers an even more flexible approach than pure Behavioral Reflection. The key advantage is that it provides a means to selectively trigger reflection, only when specific, predefined events of interest occur. Reflex uses meta-links to modify the behavior and hooksets to specify where this change should take place. A link invokes messages on a meta-object at occurrences of marked operations. The attributes of a link enable further control of the exact message to be sent to the meta-object. Reflex was implemented using bytecode transformation in Java, and is thus portable across different Java VMs. A typical use case for Reflex is the implementation of the Observer pattern [Gamma *et al.*, 1995] at the meta-level by reflecting only on those objects that are to be observed, adapting their behavior to notify their observers.

Selective Reification

Ferber [Ferber, 1989] introduced a *message reification* model of reflection where each message is an instance of a message class. Each message class can define its own

interpretation of a message send. By changing the implementation of a message class the message send semantics can be modified. In Ferber's model the sender of the message is not taken into account in the reification. Cazzola [Cazzola, 1998] extended this model by including the sender object in the message reification.

Iguana [Gowing and Cahill, 1996] takes a step forward in the meta-level architecture through dynamic reifications. Iguana offers a form of selective reification making it possible to select program elements down to individual expressions. This tool provides a fine-grained MOP which allows different object-models to coexist at the same time in the same system. It also allows dynamic changes to be applied in an object-specific manner. If an object of a given class is adapted, no other instance of that class should be affected by this change. Iguana was developed for C++ and works by placing annotations in the source code to define behavioral reflective actions.

Reifying a message send means to model as an object the event that a message has been sent to another object. Smalltalk-76 [Ingalls, 1978] reified message sends on the whole system thus impacting negatively on performance. CodA proposed a decomposition of the message send into multiple, finer-grained events while imposing this reification on the whole system. Iguana provided message send reifications not affecting the whole system.

An example of selective reification in Reflex is transparent Futures. A future is an object whose value may not yet be available as it is still being computed. Futures implemented as generic classes rather than as a built-in language construct have the disadvantage that a client of a future must explicitly request the value of the future when it is needed, as is the case in Java. A transparent future, on the other hand, could be used directly as a regular object, without the need to ask for its value. Reflex implements transparent futures at the meta-level by reifying the message reception and the object casting.

Tools like Dalang [Welch and Stroud, 1999], Reflective Java [Wu, 1998], Kava [Welch and Stroud, 2001], the ProActive MOP [Caromel *et al.*, 2001], MetaXa [Golm and Kleinöder, 1999] and Guaranà [Oliva and Buzato, 1999] are targeted specifically at controlling method invocation for Java. All of them work by manipulating bytecode.

Aspect Oriented Programming (AOP) [Kiczales *et al.*, 1997b] provides a general model for modularizing cross cutting concerns. Join points define points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (i.e., dependent on control flow). Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system. Kiczales *et al.* [Kiczales *et al.*, 1997b] claim: *"AOP is a goal, for which reflection is one powerful tool."*. Although aspects can be dynamically enabled or disabled, they are specified statically. AspectS [Hirschfeld, 2003] is a dynamic aspect system defined in the context of Smalltalk. Aspects lack an important ingredient that we were looking for, namely

they do not provide an extensible model for new reifications. The join points provided by aspect languages, mostly AspectJ-like pointcut-advice models which dominate the landscape of AO language design, are too restrictive. Events that do not naturally correspond to the boundaries of methods or field accesses cannot be easily added [Gasiunas *et al.*, 2011]. For example, if we have a temperature sensor and the event `tempChange` depends on a thread that tests for temperature changes in a sensor (Listing 2.1 lines 5–15). This event cannot be expressed with the pointcut-advice model.

```
1  class TemperatureSensor {
2     public delegate void TempChange(int newTemp);
3     public event TempChange tempChanged;
4     ...
5     public void run() {
6        int currentTemp = measureTemp();
7        while (true) {
8           int newTemp = measureTemp();
9           if (newTemp = currentTemp) {
10             if (tempChanged = null) { tempChanged(newTemp); }
11             currentTemp = newTemp;
12          }
13       sleep(100);
14       }
15    }
16    ...
17 }
```

Listing 2.1: Temperature sensor

Unanticipated Changes

Iguana/J [Redmond and Cahill, 2002; Redmond and Cahill, 2000] is the implementation of Iguana for Java. This tool enables unanticipated changes to Java applications at run-time without requiring instrumentation or restarting the application before the first use of reflection. Since the event reifications are defined in the modified VM, precise operation occurrences of interest cannot be discriminated nor can the actual communication protocol between the base and meta-level be specified. For example, a new event which reifies the execution of the garbage collector cannot be defined without modifying the VM again.

Both Iguana and Iguana/J contributed significantly to modeling the meta-level by proposing *fine-grained MOPs*. The idea of fine-grained MOPs is to allow multiple reflective object models to coexist in a given application. Nevertheless, the modified VM implementation precludes a homogeneous environment; some reifications work at the VM level while others work at the application level.

Meta-level engineering in Reflex is highly flexible but it suffers from a key limitation. Although the reflective behavior is available at run-time, the framework forces the user to anticipate the reflective needs at load time. This means that Reflex does not allow a programmer to insert new reflective behavior affecting already-loaded classes into a running application. The application has to be stopped, the reflective needs have to be specified, and then the application has to be reloaded for the reflective changes to take place.

Denker introduced Reflectivity [Denker, 2008], an implementation of the Reflex model for Smalltalk. Reflectivity targets two important problems present in the previous tools regarding behavioral reflection. These problems are anticipation and sub-method structure. Iguana/J introduced a working implementation of unanticipated partial behavioral reflection (UPBR) but suffered from portability issues. Reflex requires the user to anticipate where reflection is going to be needed.

Reflectivity provides UPBR while maintaining portability. This was achieved by using reflective methods that are dynamically compiled thus enabling unanticipated change. Persephone [Denker *et al.*, 2007] introduced a model for reflective methods and was responsible for recompiling methods that had been reflectively modified.

Reflectivity exploits the reflective structures of Smalltalk. ASTs are used as the sole representation of behavior. Reflex hooksets were removed and links were just realized as annotations to any AST node thus simplifying the Reflex model. Using AST nodes allowed Reflectivity to achieve sub-method reflection capabilities.

Nevertheless, when faced with a complex adaptation scenario links are too low-level and their management has to be specified by the user explicitly. For example, if we need to debug and halt the execution when a particular instance variable is accessed we need to find all the AST nodes in which the variable is accessed and attach a link to them. After that we realize that we also want to halt the execution when a particular method is invoked in a particular object. We require a new link that checks at runtime that the receiver of the method is the specific object, and then we attach this link to the AST method node. We have a complex adaptation scenario with several links to obtain a debugging behavior change. The semantic meaning of the set of links is lost after they are installed since there is no abstraction that states that these links belongs to the same adaptation. First, we need to find the right AST node which, if adapted with a link, will produce the required effect. Second, to remove the debugging adaptations we have to manually manage many links.

Moret *et al.* introduced Polymorphic Bytecode Instrumentation (PBI) [Moret *et al.*, 2011], a technique that enables run-time selection amongst several, possibly independent instrumentations. These instrumentations are saved and indexed by a version identifier. These versions can control the visibility of the adaptations. CodeMerger, the PBI implementation for Java, instruments the class library at build-time and all other classes at load-time, thus achieving full unanticipation is not possible.

Runtime Integration

Iguana/J was implemented using the Java Just-in Time (JIT) interface by defining a dynamic library. Instead of using annotations in the source code for specifying reflective actions Iguana/J uses a definition file. This file is compiled by a special Iguana compiler which generates dynamically the code to be executed. This technique is useful since the tool has access to the internal structures of the interpreter. However, this solution is coupled to a particular VM implementation, since the VM developers did not continue developing the JIT interface, Iguana/J does not run in more recent VMs. Reflex provides a more portable solution by transforming Java bytecode.

MetaclassTalk [Bouraqadi, 2004] extends the Smalltalk model of meta-classes by actually having meta-classes define the semantics of message lookup and instance variable access. Instead of being hard-coded in the virtual machine, occurrences of these operations are interpreted by the meta-class of the class of the currently-executing instance. A major drawback of this model is that reflection is only controlled at class boundaries, not at the level of methods or operation occurrences. This way MetaclassTalk confines the granularity of selection of behavioral elements towards purely structural elements.

Meta-level Composition

Tanter [Tanter, 2006] stated that composition of meta-objects is complex and not well supported. In CodA there is no mechanism for composition. The required changes have to be composed and placed by hand in the right meta-object. The link abstraction of Reflex and Reflectivity offers a means to compose adaptations at the bytecode and AST level, however, these approaches do not provide a mechanism for composing higher-level abstractions. CLOS-MOP provides a composition mechanism through the method combinations meta-object. Iguana/J provides a composition mechanism through the definition of MOPs. An Iguana MOP is composed of a set of meta-level events to adapt an object or class. The composition is limited to the Iguana predefined events.

Mezini [Mezini, 1997] identified that the mechanisms for incremental behavior composition do not support evolving objects at all or do not satisfactorily solve the encapsulation and name collision problems associated with them. Mezini points out that the inability of the existing approaches to uniformly handle dynamic composition and internal encapsulation is due to the lack of sufficient abstraction levels in their design. The author proposes a composition mechanism which deals with these issues by reifying a combination layer between the object and the software component that defines its behavior. A *combiner-metaobject* is associated with each evolving object to control the composition. The *adjustments* are responsible for providing mixin-like behavioral adaptations depending on the context.

Mirrors offer a first attempt to reify reflection. In this approach objects themselves do not have any reflective capability, but reflection is provided by *mirror objects* [Bracha and Ungar, 2004]. Mirrors offer a clear separation of the base level and the meta layer. However, mirrors do not specify a composition mechanism for the meta-level. This means that composing two reflective changes on an object's mirror can only be done by hand.

Ruby [Matsumoto, 2001] introduced a form of mixins [Bracha and Cook, 1990] as a building block for reusability, called modules. Modules can be applied to specific objects without modifying other instances of the class by adding or modifying state and methods. Aliasing of methods is possible to avoid name collisions, as well as removing method in the target object. However, instance or class methods cannot be removed if they are not already implemented. This follows the concept of linearization of mixins. Filters in Ruby provide a mechanism for composing behavior into preexisting methods. However, they do not provide support for specifying how method defined in modules should be composed for a single object.

PBI instrumentations are saved and indexed by a version identifier. Thus, this technique can at runtime control which adaptations are active.

Scoped Reflection

Dynamic adaptation is traditionally realized with the help of activation conditions evaluated at runtime to decide which parts of the system should be adapted. Even very dynamic approaches like unanticipated partial behavioral reflection [Röthlisberger *et al.*, 2008] only shift the time when conditions are added from load time to runtime. This flexibility allows the programmer to reduce the number of checks performed or to remove unneeded ones at runtime. Yet, it does not solve the real problem: foreseeing the parts of the system where checks are to be added. This is not always possible, since in many cases the system is unfamiliar to the developer, or system libraries are also under analysis. Dynamically scoped adaptations were introduced to deal with these situations. A dynamic extent defines a dynamic scope by providing a piece of code to be executed. As the code is executed adaptations are installed, and propagated under certain conditions.

Tanter [Tanter, 2009] formalized dynamically scoped adaptations in terms of (i) the dynamic extent, (ii) the propagation function, (iii) activation conditions and (iv) the adaptations to be applied. The propagation function defines how the adaptations should be propagated in the dynamic extent. The use of activation conditions can further control the application of the adaptation during the dynamic scope.

However, a key problem with previous techniques is that the scope cannot be modified at runtime. The propagation function, the activation condition and the adaptations, once defined, cannot change. Recently, Moret *et al.* introduced Polymorphic

Bytecode Instrumentation (PBI) [Moret *et al.*, 2011], a technique that enables runtime selection amongst several, possibly independent instrumentations. These instrumentations are saved and indexed by a version identifier. These versions control the visibility of the adaptations. For example, if we applied two different analyses, like feature analysis and profiling, over the same system we would like both adaptations not to interfere with each other. This means, that feature analysis should not take into account extra behavior introduced by the profiling adaptation and vice versa. If a method is adapted by both analyses then there are two different versions of the method, one adapted with feature analysis behavior and the other with profiling behavior. When executed each thread of execution will have an index that defines which version of each method should be selected, thus avoiding conflict.

A key issue in dynamic adaptation is to control the scope and visibility of the adaptations. Previous approaches are capable of scoping changes at the class level, for a single object, thread-locally or globally [Tanter, 2007]. For instance, CaesarJ [Aracic *et al.*, 2006] supports per-thread aspect deployment, where an aspect instance can see all join points produced in the dynamic extent of an execution of block. A similar mechanism can be found in AspectScheme [Dutchyn *et al.*, 2006] and AspectS [Hirschfeld and Costanza, 2006]. Other approaches control the scope of aspects to be deployed on specific objects [Aracic *et al.*, 2006; Rajan and Sullivan, 2003], or globally [Aracic *et al.*, 2006; Rajan and Sullivan, 2003; Hirschfeld, 2003; Suvée *et al.*, 2003] Deploying an aspect on a specific object means that if we have a class with two instances we will only adapt the specific object. Global deployment will adapt all instances of a class.

Object-Specific Reflection

Object-specific reflection offers an approximation to object-centric reflection. Previous reflective mechanisms provide reflective models where object-specific capabilities are just one technique amongst many. Object-specific reflection enables reflection on specific objects, but may not avoid users reflecting on static abstractions. Object-centric reflection avoids static abstractions altogether. Several tools present various mechanisms to achieve it.

CLOS-MOP, for example, has six kinds of meta-objects: classes, slots, generic functions, methods, specializers and method combinations. These concepts relate to how objects are described by users, not how they are run by computers. Partial reflection can be achieved by specializing the meta-class generic functions for a specific meta-object class [Attardi *et al.*, 1989]. However, CLOS-MOP does not support object-specific method invocation reification in a scalable way, as McAffer [McAffer, 1995a] pointed out. Meta-level changes have to be described in methods. Methods simply do not provide the infrastructure and abstraction necessary for describing more than very simple behaviours. They do not directly support reuse, combination or composition. They are not suitable units of encapsulation for engineering the meta-level.

Self [Ungar and Smith, 1987] is a prototype-based language that follows the concepts introduced by Lieberman [Lieberman, 1986]. In Self there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Object-specific behavior comes naturally to this model of reflection. From a reflective point of view Self concentrates on structural reflection and supports behavioral reflection to a certain extent. For example, method lookup is reified but achieving a full operational decomposition is not straightforward. Thus, Self has a strong object-oriented reflection mechanism but the lack of an operational decomposition mechanism prevents a fully object-centric approach.

Iguana and Iguana/J provide object-specific behavior as a core feature. Reflex and Reflectivity adaptations can be attached only to operations in the source code representation. There is no mechanism to attach a reflective adaptation to a class, instance variable or object. Object-specific adaptations can be achieved by introducing object-related conditions in the adaptation. However, this is not an object-centric approach since adaptations are not only targeted at objects and object-specific adaptations require conditions introduced by the user. For example, if we want to debug the access to a particular instance variable of a specific object, we cannot apply a link to the object. We need to apply a link on all AST nodes which access the variable at the object's class level and introduce a special condition. This condition checks at runtime that we should only halt the execution when the `this` variable is equals to the target object.

MetaclassTalk [Bouraqadi, 2004] extends the Smalltalk model of meta-classes by actually having meta-classes define the semantics of message lookup and instance variable access. Instead of being hard-coded in the virtual machine, occurrences of these operations are interpreted by the meta-class of the class of the currently-executing instance. A major drawback of this model is that reflection is only controlled at class boundaries, not at the level of methods or operation occurrences. This way MetaclassTalk confines the granularity of selection of behavioral elements towards purely structural elements. Objects are not the main target of MetaclassTalk reflective changes thus this approach is not object-centric.

Cola [Piumarta and Warth, 2006] implements an open object model for experimenting with different programming paradigms. Though this model is quite powerful, the abstractions that it provides are based on lookup tables. The user can deal only with these abstractions and no higher-level abstractions are provided to leverage the level of expressiveness.

Dynamically-scoped aspects present different tools supporting object-specific adaptations. Following the idea of per-object meta-objects Rajan and Sullivan [Rajan and Sullivan, 2003] propose per-object aspects. An aspect deployed on a single object only sees the join points produced by this object. CaesarJ [Aracic *et al.*, 2006] provides *deploy blocks* which restrict behavioral adaptations to take place only within the dynamic extent of the block. PBI can scope changes to specific objects, however

this is seen as a particular case of scoping and not as a core mechanism thus is not a pure object-centric approach.

For example, traditional debuggers are focused on the execution stack. The developer identifies which parts of the source code are of interest and sets breakpoints accordingly. The software then runs until a breakpoint is reached, and the developer can then inspect and interact with the code and entities in the scope of the breakpoint. Unfortunately this process is ill-matched to typical development tasks. Breakpoints are set purely with respect to static abstractions, rather than to specific objects of the running system. It has been proved that when debugging the developer ask question related to the runtime abstractions, like what is the value of an arguments at runtime [Sillito *et al.*, 2006]. Due to this developers are less efficient because they have to reflect on classes and methods to reach object-specific information.

When we look deeper into how languages implement reflective applications we observe a chronic pattern to move away from the runtime abstractions towards static ones. Even though, as we have seen, some reflection approaches are capable of providing object-specific adaptations. What is missing is to steer the user to think in terms of object and their runtime behavior by having a reflective system centered on object reflection.

2.3.2 Summary

There has been extensive work on partial reflection, selective reifications, unanticipated changes, runtime integration, meta-level composition, scoped reflection and object-specific reflection suggesting that they are key requirements in achieving a compelling approach to reflection.

We have presented various techniques which provide object-specific behavior: aspects deployed on specific objects [Aracic *et al.*, 2006; Rajan and Sullivan, 2003], CaesarJ [Aracic *et al.*, 2006], PBI [Moret *et al.*, 2011], *etc.* Each of these approaches solve some but not all of the presented reflection requirements. Moreover, object-specific reflection is stressed as an important feature [Rajan and Sullivan, 2003; Aracic *et al.*, 2006; Moret *et al.*, 2011; Gasiunas *et al.*, 2011] but at the same time these approaches present heterogeneous mechanisms to achieve other adaptations. Due to this object-specific reflection is not modeled to be the central reflection mechanism, thus, we observe the object paradox in many of the reflection applications like debugging, feature analysis and profiling.

Traditional reflective models are focused around static source artifacts and only provide very limited access to the dynamic parts at runtime. The research question we pose is: What kind of reflective model do we need to avoid the object paradox while supporting the requirements of reflection under a unified and uniform solution?

Chapter 3

Object-Centric Reflection

In this chapter we explain the object-centric reflective model of explicit meta-objects. As we have seen, various approaches to reflection address different needs, but the object-paradox is not their main concern. We seek to develop a new model of reflection that addresses the object paradox and the standard requirements identified by past approaches while integrating and unifying the essential features of these existing approaches. The key idea behind object-centric reflection is to provide object-specific capabilities as the central reflective mechanism.

3.1 Object-Centric Reflection in a Nutshell

The key difference with previous approaches is that instead of adding object-specific capabilities for reflective adaptation to an existing reflective framework, we adopt object-specific adaptations as the core of our approach. For this reason we refer to it as *object-centric reflection*. With object-centric reflection we can: (i) avoid the object paradox, (ii) provide a unified approach to meta-level engineering, and (iii) simplify the reflection model.

Meta-objects are responsible for defining the structure and the behavior of specific objects. Any object can be bound to one or more meta-objects, and various meta-objects can adapt the behavior or structure of various parts of the same object. When a meta-object is bound to an object, it forms part of the meta-level description of the object. For example, we can define a method wrapper for a specific object. When the meta-object is unbound from an object then this object no longer responds to the meta-description modeled by that meta-object.

The meta-object abstraction is unique in the sense that every meta-level abstraction is expressed in terms of a meta-object. Every meta-object instantiation is an instance of a meta-object or a specialization. A meta-object is also an object, and thus, it is also possible to create meta-meta objects to control the meta-objects. We can model a class with a meta-object defining the methods of potential instances. Instance creation is performed by sending the message `new` to the class. Thus we can define and bind another meta-object to our class abstraction where class-side methods such as

`new` are defined. This meta-object models a meta-class, also known as meta-meta-object (the meta-object of a meta-object).

A structural meta-object is responsible for modeling the structures of a program. An object-oriented program's structures are classes, traits, methods, message send AST nodes, *etc.* Structural meta-objects deal with the definition of meta-level structural reifications. For example, adding a method to a particular object. How and when they are introduced at run time is the job of the behavioral meta-object.

A behavioral meta-object is responsible for modeling the dynamic representation of a program. Examples of such reifications are: the message send, the method lookup, or the object creation. We can build a profiler by applying a behavioral meta object that increments a counter everytime a message is sent. A different counter can exist for each adapted object and method.

The relationship between object and meta-object is controlled. Compound meta-objects enable safe meta-level composition. They are reified to avoid potential conflicts between meta-objects and to manage meta-object adaptation performed on objects. These meta-objects are composed of multiple meta-objects.

Compound meta-objects encapsulate the complexity of dealing with multiple meta-objects at the same time. When an object is bound to more than one meta-object then there is a single compound meta-object modeling the composition between these meta-objects. Consider two behavioral meta-objects, one modeling a profiler with a counter of message sends, and the other a test coverage analysis adaptation registering the executed methods. If we bind these two meta-object to the same object the meta-objects are composed and they adapt the object. If we request the meta-object of the bound object we receive a compound meta-object representing the composition. This behavior is managed transparently from the user unless a particular composition requirement needs to be fulfilled. We present examples of various composition mechanisms in Chapter 5 and Chapter 6.

3.2 Meta-objects

Object-centric reflection supports three kinds of explicit meta-objects, which eventually can be extended to reify new meta-level abstractions as we show later in this dissertation:

- `StructuralMetaObject` and `BehavioralMetaObject` reify respectively structural and behavioral reflective capabilities.
- `CompoundMetaObject` reifies the composition of meta-objects.

3.2.1 Structural Meta-object

A `StructuralMetaObject` acts on the basic structural units of an object-oriented language which are messages, objects and objects' states. The responsibilities of a `StructuralMetaObject` are:

- *Adding a method.* A new method is added to the object.
- *Removing a method.* The adapted object will not understand a particular message anymore.
- *Replacing a method.* The method will have another behavior. Either explicit source code or a closure can be provided.
- *Adding state.* The addition of new state to an object allows the user to add methods that use that state.
- *Removing state.* Specific state is removed.

Structural meta-objects deal with the definition of meta-level structural reifications. How and when they are introduced at run-time is the job of the `BehavioralMetaObject`.

3.2.2 Behavioral Meta-object

A `BehavioralMetaObject` reifies the meta-object responsible for modeling the dynamic representation of a program. Examples of such reifications are: the message send, the method lookup, or the object creation. This abstraction corresponds to the work done in Iguana and later used by McAffer in CodA. As McAffer pointed out, the system is modeled as a set of operations whose occurrences *"can be thought of as events which are required for object execution"* [McAffer, 1995b].

To dynamically adapt the behavior of an object we need to describe what we would like to do and when. To specify what we would like to apply, we delegate the responsibility of managing an event to a specific meta-object. We specify when it should be adapted by using a computational event in the execution of a program, *e.g.*, sending a message.

A set of canonical events models the basic operations known as *dynamic reification categories*. The dynamic reification categories are: message send, message receive, state read, and state write. These are not the only reifications possible; new dynamic reifications can be defined, the only requirement being to specify when they should be triggered. For example, *entering a synchronized block* can be modeled by a meta-object that adapts the points in an object where, depending on the implementation of the language, a synchronized block is accessed. In Smalltalk this is done using the method `critical: aBlock`.

We selected the above categories following the Iguana approach. Iguana proposes seven canonical reification categories, some of which can be defined in terms of the others. For example, the object creation event can be expressed as a message send extension, since an object is created when the creation message is sent to a class (the same applies for object deletion). With these basic categories we can adapt an object's behavior from an operation decomposition point of view.

3.2.3 Compound Meta-object

A meta-object can be bound to an object, and unbound. A compound meta-object manages the composition between meta-objects with two mechanisms:

- *Order.* The order in which the adaptation expressed by meta-objects is applied might be meaningful in certain cases. By default a meta-object is appended at the end when composed with a compound meta-object. Otherwise, its position has to be explicitly stated by using different methods. When two compound meta-objects are composed then either one of them takes precedence over the other or the user has to order them again.

- *Conflicts.* When a new meta-object is added to a compound meta-object, multiple user-defined conflict validations are evaluated. By default the compound meta-object checks that behavior that is expected by a meta-object is not removed and modified by other meta-objects. Furthermore, conflict validation rules can be added to the compound meta-meta object to check for potential conflicts when new meta-objects are added. Conflict validation rules model the evaluation of specific behavior to check for adaptation conflicts. When a conflict is detected between a compound meta-object and a meta-object, this meta-object throws an exception modeling the error. A meta-object can catch this exception to either fix the conflict or ignore it. The exception handler's default behavior is to reject the composition. For example, two different meta-objects try to add the same method `asString` to a single object. Since there is no way of deciding which of the two definitions should be used the composition of the two meta-objects is rejected.

3.2.4 Scoping Meta-object adaptations

Prisma is an approach to dynamically adapt running software systems to support various forms of dynamic analysis. Prisma uses object-centric reflection to adapt the behavior of objects at run time. *Execution is modeled as a sequence of events* that trigger the reflective meta-objects. Prisma explicitly *reifies execution runs* to manage the adaptation process. Dedicated *propagation meta-objects* assume responsibility for propagating adaptations to objects accessed within a given run. Adaptations are

scoped to a particular run, so multiple adaptations can be installed without risk of interference. *Installation and deinstallation are decoupled*, so adaptations can be retained for long-lived analyses.

Execution Reification. In many programming languages it is possible to reify abstractions such as activation records, execution contexts, and even the execution stack, but the concept of an *execution run* remains implicit. Prisma models execution runs explicitly to scope adaptations to a specific set of objects reachable from a particular starting point. An execution run represents a live scope within which adaptive reflective changes take place.

An *execution* is composed of a set of *meta-objects*, each of which adapts a number of *bound objects*. Since a meta-object is an object, it can also be adapted by meta-objects. Meta-objects can be structural or behavioral. An execution models a dynamic scope whose starting point is an expression defining a dynamic extent.

A dedicated *propagation meta-object* is responsible for propagating adaptations to the dynamic extent of an execution run. When an execution run is started the first object, *i.e.*, the one receiving the first message, is adapted with the meta-objects composing the execution. One of these meta-objects is the propagation meta-object, which adapts an object so that every method call to another object causes the execution's meta-objects to be applied to that other object. An *activation condition* can be provided to restrict which objects the adaptation should be applied to.

Execution Scoping. When an object is adapted within a particular execution run this adaptation only affects other objects in the same run. When a meta-object adapts a method of an object under a specific execution run the method is copied and the adaptation is applied to that copy. As a consequence, there can be multiple versions of the same method for a given object depending on the number of scoped executions. The meta-object is responsible for managing the different method versions. When the adapted method is invoked under a particular execution run, (i) the invoked object delegates the execution of the method to the meta-object, (ii) the meta-object obtains the identity of the current execution, and (iii) with that identity it selects the version of the method to be executed. If there is no enclosing adaptive execution then the normal method lookup is used. For example, feature analysis requires the developer to adapt beforehand all the classes that he wants to be taken into account by the analysis. This is not always possible because sometimes the extension of the system is unknown. Execution scoping allows the developer to adapt objects as they are reached by the execution. Moreover, traditional feature analysis forces the developer to exercise a single feature at a time. For example, if we have two users trying to exercise at the same time the *printing* and *login* features respectively the adapted code when executed cannot easily discerns which feature is being exercised. Execution scoping allows the developer to have various execution contexts where different developers are exercising different features.

3.3 Meta-object Definition

Meta-objects define the structure and behavior of objects. The meta-level is composed of a set of meta-objects that can eventually be described by other meta-objects. This renders a complex structure of relationships between objects and meta-objects. Traditional reflective models tend to simplify this structure by using classes and meta-classes or by only allowing a single meta-object per object. Removing this limitations leaves us with the complexity of managing meta-objects. To control this complexity we propose a very simple process for defining objects and meta-objects.

There are three cases for defining what should be done to model the meta-level.

1. If we have the object and the meta-object then we just bind them.
2. If we do not have the meta-object then we create it and bind it to the object.
3. If we have neither the object nor the meta-object then we need to create the object first by locating the meta-object that can create it.

As an example, let us consider a logger which logs the execution of an object's particular method. We know the object to be adapted which is the method. Thus, we only need to create the meta-object and to bind it to the object. This is a case 2 scenario.

Let us consider reifying message sends, which means that when an object sends a message to another object an event modeling this situation will be explicitly created. This is a case 3 example since we do not have the object (message send event) and we do not have the meta-object responsible for the reification. Thus we need to create both the object and the meta-object.

3.4 Unification of Reflection

Object-centric reflection does not only fulfill the reflection requirements, it also does it under a unified meta-model. Next we discuss how object-specific meta-objects simplify the implementation and unification of the reflection requirements.

Partial Reflection. Meta-objects make reflective changes available only in selected places where needed. There is no need to modify the whole system.

Selective Reifications. Meta-objects can be used to model new reifications, for example in the case of Prisma the execution run. Moreover, when adapting a particular object the meta-object can selectively define which data should be reified at runtime, for example, the sender and the receiver of a particular message, the arguments of the message, the continuation, *etc.*.

Unanticipated Changes. Meta-objects can be applied to any object in the system at runtime without the need of previously stating that this adaptation will take place.

Runtime Integration. Low-level meta-objects are responsible for the runtime integration. This meta-object abstracts away from the lower level implementation detail thus providing a reusable model. Thus meta-objects live at the same level as the application.

Meta-level Composition. Compound meta-objects provide the semantics for composing meta-objects. Adaptations can be composed at runtime according with the requirements of the user.

Scoped Reflection. Meta-objects can scope their structural and behavioral adaptations. Prisma allows adaptations to be scoped to particular executions as the system is running.

As we can see the reflective requirements a merged with the meta-object architecture.

3.5 Object Paradox

Object-centric reflection avoids the object paradox by making object specific adaptation the central reflection mechanism. In the remainder of this dissertation we show how this simple change in perspective forces the developer to first think about the objects rather than the static representation of the running system. Moreover, we demonstrate how developers using object-centric reflection become more efficient due to reduction in the gap between the user's object-specific questions and the reflection mechanism features.

This approach does not at all prevent the user to reflect on the static representation. On the contrary it allows the user to reflect directly on the objects representing the static representations. There has been some research on the questions a developer asks when analyzing and developing a software system [Sillito *et al.*, 2006; Sillito *et al.*, 2008]. The developers' questions are mainly centered on specific objects and particular interactions at runtime. There is a gap between the developer questions and what traditional tools provide. Traditional tools and reflection techniques partially cover these requirements without a unified approach. Object-centric reflection fills this gap providing a unified approach to organize the meta-level while fulfilling the reflection requirements.

In the next chapter we present an object-centric reflection implementation and validate the claims stated in this chapter through a series of examples.

Chapter 4

Bifröst

In this chapter we present Bifröst[1], our object-centric reflection approach implemented in Pharo[2] Smalltalk [Black *et al.*, 2009]. We also present several examples of how Bifröst is used and validate how Bifröst solves the object paradox and how the reflection requirements are fulfilled.

4.1 Meta-objects

We will introduce the various Bifröst meta-objects and how they support object centric-reflection.

Bifröst has four kinds of explicit meta-objects, which eventually can be extended to reify new meta-level abstractions as we will see later in this dissertation:

- `StructuralMetaObject` and `BehavioralMetaObject` reify respectively structural and behavioral reflective capabilities (see Figure 4.1).
- `CompoundMetaObject` reifies the composition of meta-objects.
- `LowLevelMetaObject` reifies meta-objects that are responsible for adapting low level structures like AST nodes.

4.1.1 Structural Meta-object

A `StructuralMetaObject` acts on the basic structural units of an object-oriented language which are messages, objects and objects' states. The responsibilities of a `StructuralMetaObject` are:

1 In Norse mythology, Bifröst is the burning rainbow bridge between the worldly realm and the heavens.
2 `http://www.pharo-project.org/`

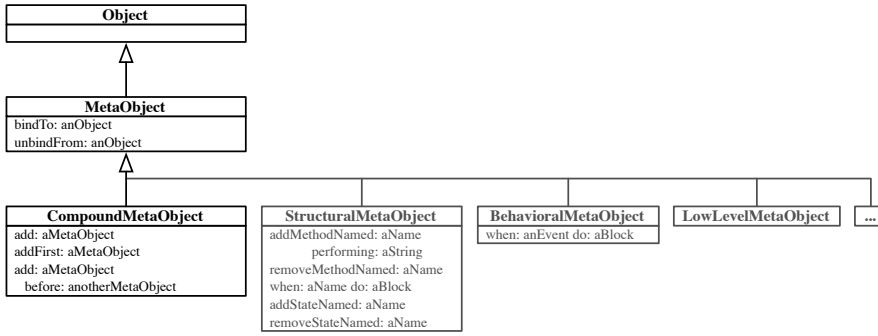

Figure 4.1: Meta-Objects class diagram with methods denoted in Smalltalk.

- *Adding a method.* A new method is added to the object. A name and the source code is provided. When the object receives the corresponding message it executes the compiled source code. The source code compilation is performed when the object is associated with a meta-object. If there is a compilation error the meta-object association is rolled back.
- *Removing a method.* The adapted object will not understand a particular message anymore.
- *Replacing a method.* The method will have another behavior. Either explicit source code or a closure can be provided.
- *Adding state.* The addition of new state to an object allows the user to add methods that use that state.
- *Removing state.* Specific state is removed.

All these adaptations are not permanent, the user can undo an adaptation at any time. Structural meta-objects deal with the definition of meta-level structural reifications. How and when they are introduced at run-time is the job of the `BehavioralMetaObject`.

4.1.2 Behavioral Meta-object

We selected the above categories following the Iguana approach. Iguana proposes seven canonical reification categories, some of which can be defined in terms of the others. For example, the object creation event can be expressed as a message send extension, since an object is created when the message `basicNew` is sent to a class (the same applies for object deletion). With these basic categories we can adapt an object's behavior from an operation decomposition point of view.

The method `when:do:` specifies that when a particular meta-event happens the particular behavior in the block should be executed.

On the other hand, the method `StructuralMetaObject>>when:do:` replaces the body of a particular method by another method body. As a consequence, the behavior is also changed but, as we have seen, changes to the structure of a system can affect its behavior as well.

4.1.3 Compound Meta-object

Talents [Ressia *et al.*, 2011] are dynamically composable units of reuse built on top of Bifröst. Talents can be composed. The composition order is irrelevant so conflicting talents must be explicitly disambiguated. Composition operators are used to solve conflicting compositions *i.e.*, aliasing the names of the adapted methods and deleting particular methods.

For example, streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams may be either readable, writeable or both; they can also be binary or character-based; and we can have memory streams, socket streams, database streams, or file streams. Dynamically composing the right combination of streams required is key for avoiding an explosion of classes due to all potential combinations. `WriteStreamTalent` adds the methods for writing to a stream, *i.e.*, `nextPut:` and `nextPutAll:`. `ReadStreamTalent` adds the methods to read from a stream, *i.e.*, `next` and `next:`. Composing these two talents delivers a readable and writable talent. But if we compose this talent with a `BinaryReadStream-Talent` which defines `nextPut:` in a different way, then we need to decide how the combination should be performed. We could choose to keep the implementation of one of the talents over the other by removing a method from the composition. Or we could keep both methods by aliasing the method `nextPut:` from one of the talents to avoid a conflict when composing them.

4.1.4 Low-level Meta-object

Low-level meta-objects are responsible for providing the low-level mechanisms needed to modify the system behavior.

The design of Bifröst can be understood as an evolution of Reflectivity, which in turn was conceived as an extension of the Reflex model of *Partial Behavioral Reflection*. In the Reflex model, and in Reflectivity, links are attached to AST nodes to modify their associated behavior. Links are hard to manage, for example, to produce a debugging adaptation you need to find the right AST nodes where to place the link or links. The semantics of the adaptation are distributed across multiple links that have no connection between each other. Meta-objects offer a solution to this problem by providing a higher level adaptation abstraction.

The simplest building block provided by our approach is the low-level meta-object. Bifröst provides low-level meta-objects that adapt AST nodes. The AST meta-objects

are responsible for changing the behavior of an AST node. For example, a simple message send adaptation can be achieved by attaching a low-level meta-object to a message send AST node. These meta-objects are used by the compiler to adapt the compilation process and change the normal behavior of a specific instruction.

4.2 Bifröst Exemplified

In this section we demonstrate how our approach supports the requirements of reflection (partial reflection, selective reifications, unanticipated changes, runtime integration, meta-level composition and scoped reflection) by means of several examples. Table 4.1 provides an overview of the examples and how they cover the reflection requirements. The numbered scenarios depicted in every example are related to the different meta-object cases presented in Section 3.3.

Section	Example	Partial Reflection	Selective Reification	Unanticipated Changes	Runtime Integration	Meta-level Composition	Scoped Reflection
4.2.1	Profiling (Scenario 2)	●	●	●	●	●	○
4.2.2	Traits (Scenario 2)	●	○	●	●	○	○
4.2.3	Delegates (Scenario 2)	●	●	●	●	○	○
4.2.4	Prototypes (Scenario 3)	●	●	●	●	●	○
4.2.5	Live Feature Analysis (Scenario 2)	●	●	●	●	●	○
4.2.5	Scoped Live Feature Analysis (Scenario 2)	●	●	●	●	●	●

Table 4.1: Bifröst coverage over reflection requirements.

Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An invocation of a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. A method with no arguments looks like `receiver method`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret `^` returns the result of the following expression. In Smalltalk objects interact by sending messages to each other. A method invocation is therefore called *a message send*, and an object's method is called when a *message is received*.

4.2.1 Profiling (Scenario 2)

Profiling is a common example of object adaptation. We demonstrate how to build a simple profiler with Bifröst meta-objects. Listing 4.1 demonstrates how to count instance variable read accesses on a particular object.

```
variableReadProfiler := BehavioralMetaObject new.
variableReadProfiler
   whenStateIsReadDo: [ counter := counter + 1 ].
variableReadProfiler bindTo: anObject.
```

Listing 4.1: Variable read counter.

The reification is scoped to the bound objects only. Accessing state of another instance of the same or a different class does not increase the counter.

When profiling application code we require that no external stimulus disturb the measurements. When binding a profiling meta-object to an object we need to be sure no other meta-object is already adapting the object since this would affect our measurements. By using reflection we can detect that there is already another meta-object present. The granularity of the validation depends on the granularity of the meta-objects. If there is a coverage meta-object adapting a single method of an object it might happen that the profiling meta-object has no interest in that method so the composition can take place.

4.2.2 Traits (Scenario 2)

A trait [Ducasse *et al.*, 2006b] is a composable unit of behavior that can be shared among classes. If several classes share a trait then all objects that are instances of these classes are able to understand the messages defined in the trait. In this example, we demonstrate how we can introduce traits to objects, *i.e.*, running instances. For the sake of simplicity we describe only a meaningful subset of the overall traits mechanisms, namely: definition of a trait, addition of a trait to an object, composition of traits and conflict resolution. However, we take it a step further in another direction and show how we can build traits that are shareable between individual *objects*, not just classes. This example shows a first attempt at developing talents, Chapter 5 presents the details of the full implementation of dynamic composable units of reuse.

Let us assume that we have a financial system and that we want all financial instruments to share the same behavior. For example, suppose we want to provide a common implementation for the recalculate feature. Furthermore, we do not want to impose a common superclass on all financial instruments to introduce this feature, but instead keep the possibility to assign the feature dynamically to a financial instrument. We can fulfill these needs by defining the feature as a trait, however

if the host language does not provide traits we cannot introduce this feature as we would like. Bifröst provides a way to define dynamically the trait abstraction by adapting the language model.

```
1  financialInstrumentTrait := StructuralMetaObject new.
2  financialInstrumentTrait
3      addMethodNamed: #recalculate
4      performing: 'recalculate
5                      self recalculateTaxes.
6                      self recalculateDates'.
7  bond := Bond new.
8  loan := Loan new.
9  financialInstrumentTrait bindTo: bond.
10 financialInstrumentTrait bindTo: loan.
```

Listing 4.2: Building the trait abstraction with structural meta-objects.

First, we introduce the trait abstraction itself as a structural meta-object (Line 1). Then we define the message `recalculate` (Line 2–6) for this trait, its behavior being to recalculate taxes and dates. By using the existing class abstraction defined with meta-objects we instantiate two financial instruments (Lines 7–8). Finally, we associate the trait as the meta-object to both objects thus making them capable of answering the message `recalculate`.

A trait is defined as a `StructuralMetaObject`. However, by definition, traits should not have state. To achieve this we need to remove the possibility of adding state in the trait structural meta-object.

```
1  traitBehavior := StructuralMetaObject new.
2  traitBehavior removedMethodNamed: #addStateNamed:.
3  traitBehavior removedMethodNamed: #removeStateNamed:.
4  traitBehavior bindTo: financialInstrumentTrait.
```

Listing 4.3: Making traits stateless.

We first define another structural meta-object called `traitBehavior` (Line 1). This abstraction has the responsibility of defining which are the messages a trait meta-object is capable of answering. In Lines 2–3 both state-related messages are removed from the trait behavior definition. Finally, in Line 4 the `traitBehavior` is set as the meta-object of the trait meta-object defining its responsibilities. The semantics of `bindTo:` dictate that when a meta-object is bound to an object with a preexisting meta-object then a composition is executed. Objects can only have one meta-object, calling `bindTo:` does not replace the object's meta-object.

By binding meta-objects to meta-objects Bifröst can change itself uniformly. The system is not biased towards any particular reflection model.

Let us now consider the definition of another financial trait which has a conflict with the `financialInstrumentTrait`.

```
1 taxingInstrumentTrait := StructuralMetaObject new.
2 taxingInstrumentTrait
3     addMethodNamed: #recalculate
4     performing: 'recalculate
5               self recalculateTaxes'.
6 bond := Bond new.
7 financialInstrumentTrait bindTo: bond.
8 taxingInstrumentTrait bindTo: bond.
```

Listing 4.4: Building a conflicting trait abstraction.

In Listing 4.4 we define a trait which adds the method `recalculate` with a different implementation. In lines 7 and 8 we are binding the bond object to the two traits. The binding in line 8 will throw an exception stating that the adaptation in `taxingInstrumentTrait` has a conflict with a previous meta-object adaptation.

A central mechanism of traits is conflict resolution. With Bifröst the user can also provide a different behavior than the default conflict handler.

```
1  taxingInstrumentTrait := StructuralMetaObject new.
2  taxingInstrumentTrait
3      addMethodNamed: #recalculate
4      performing: 'recalculate
5                self recalculateTaxes'.
6  traitComposition := CompoundMetaObject new.
7  traitComposition
8     when: InvalidAddMethodException
9     do: [ :exception | exception
10              compoundMetaObject mergeLastConflict ].
11 bond := Bond new.
12 financialInstrumentTrait bindTo: bond.
13 taxingInstrumentTrait bindTo: bond.
```

Listing 4.5: Building a compound trait conflict manager.

In Listing 4.5 we are defining a trait that adds the method `recalculate` and a compound meta-object. This meta-object defines that when there is an error when adding a method to an object a different behavior from the default should be executed. In this case the handler in line 9–10 commands the compound meta-object to merge the last conflict. The compound meta-object has specific actions when dealing with conflicts; here it will merge the methods.

This example shows how Bifröst supports addapting an object without anticipation at run-time. We can also observe how dynamic traits are composed through the meta-object definitions.

4.2.3 Delegates (Scenario 2)

The method-lookup reification defines the process that specifies which method should be executed when an object receives a message. Most languages, including Java and Smalltalk, do not reify method lookup. Most class-based languages impose a method lookup that follows the class hierarchy and is typically hardcoded into the execution machinery.

We introduce *delegates* [Lieberman, 1986; Stein, 1987] as an example of avoiding class-based method lookup. Objects that have a delegate will be able to forward messages not understood by the receiver to another object, effectively changing the traditional lookup.

```
1  delegateStructure := StructuralMetaObject new.
2  delegateStructure addStateNamed: #delegate.
3
4  delegateForwarder := BehavioralMetaObject new.
5  delegateForwarder
6      when: MessageNotUnderstood new
7      do: [ :receiver :selector :arguments |
8          receiver delegate
9              perform: selector
10             withArguments: arguments ].
11
12 delegateStructure bindTo: anObject.
13 delegateForwarder bindTo: anObject.
```

Listing 4.6: Reifying method lookup with structural and behavioral meta-objects.

First we define a structural change, adding the variable `delegate` to the bound meta-objects (Lines 1–2). It will hold the object where messages are sent to if the receiver cannot handle them. Next we define the behavior with a behavioral change (Lines 4–10). Whenever a message is not understood (not implemented) by the receiver, the code block (Lines 7–10) is executed. It first asks the receiver for its delegate, by calling the accessor that was created by the structural meta-object. Then it invokes the same method with the same arguments on the delegate object. The last two lines (Lines 12–13) bind the two meta-objects to `anObject`. Infinite regression can happen if the object's delegate is the object itself.

This example shows how Bifröst supports partial reflection by adapting a single object with delegates. Receiver, selector and arguments are selectively reified in the behavioral event. The adaptation is achieved without anticipation and at runtime.

4.2.4 Prototypes (Scenario 3)

Lieberman [Lieberman, 1986] introduced the idea of using the prototype abstraction to better model the evolution of concepts, and thus the evolution of abstractions. Modeling with prototypes works by cloning objects from other prototypical objects. The prototype behavior and state is copied to the cloned objects. The behavior and the state of every object can be modified to model new abstractions. Any object can be a prototype.

```
1  prototypeMetaObject := CompoundMetaObject new.
2
3  prototypeStructure := StructuralMetaObject new.
4  prototypeStructure
5      addMethodNamed: #clone
6      performing: 'clone ^ Object new metaObject: prototypeMetaObject; prototype:
        self'.
7  prototypeStructure addStateNamed: #prototype.
8  prototypeStructure
9      replaceMethodNamed: #addMethodNamed:performing:
10     performing: 'addMethodNamed: aSelector performing: aString prototype
        addMethodNamed: aSelector performing: aString'.
11
12 prototypeBehavior := BehavioralMetaObject new.
13 prototypeBehavior
14     when: MessageReceived new
15     do: [ :receiver :selector :arguments |
16         (receiver respondsTo: selector)
17             ifTrue: [ self perform: selector withArguments: arguments ].
18             ifFalse: [ receiver prototype
19                       perform: selector
20                       withArguments: arguments ] ].
21
22 prototypeMetaObject add: prototypeStructure.
23 prototypeMetaObject add: prototypeBehavior.
```

Listing 4.7: Building a prototype object model.

The prototype meta-object is composed of a structural meta-object (Lines 3–10) and a behavioral meta-object (Lines 12–18) as shown in Listing 4.7. The structural meta-object defines the `clone` message (Line 4–6). This message creates an empty object and then sets its meta-object to the single prototype meta-object. In Line 7 the `prototype` instance variable is added to keep track of the cloning chain. Prototypes can add behavior and state to themselves. We therefore use a structural meta-object to model prototypes. To adapt the actual prototype we also have to change the default meta-object behavior. In Lines 8–10 we can see that the behavior of the structural meta-object is modified to delegate to the default prototype meta-object the

addition, deletion and replacement of behavior and state. For simplicity we only show the method addition example.

Lines 12–20 define the behavioral meta-object. The *message received* reification is used to adapt the behavior of the object. When this event occurs, if the receiver implements the message then it handles it, otherwise the message is delegated to the receiver prototype. Finally, a compound meta-object is created (Lines 22–23) with the behavioral and the structural meta-objects previously defined.

```
1 pen := StructuralMetaObject new.
2 prototypeMetaObject bindTo: pen.
3 pen prototype: prototypeMetaObject.
4
5 pen addMethodNamed: #color performing: 'color ^ Color red'
6
7 pencil := pen clone.
8 pencil addMethodNamed: #hasRubber performing: 'hasRubber ^ true'
```

Listing 4.8: A prototype pen use case.

Listing 4.8 presents a user-case for the prototype model. The objective is to model a 'pen' and 'pencil' using prototypes. Lines 1–3 define the 'pen' prototype. In Line 5 the method color is added to the 'pen' prototype answering red. Then the pencil prototype is created by cloning the pen prototype (Line 7). The pencil knows how to answer the color message by delegation to the original prototype. Line 8 adds a new method to the 'pencil' prototype which is only relevant to the pencil thus the 'pen' does not know it.

This example shows how to compose meta-level objects with compound meta-objects. Dynamically defining prototypes proves that Bifröst is capable of defining new reflective models that can coexist with other reflective models; consequentially this approach is not biased to a particular reflective model.

4.2.5 Live Feature Analysis (Scenario 2)

Feature Analysis determines which software entities in a complex software system support which end-user features. Traditional approaches to feature analysis establish this correspondence by exercising features to generate a trace of run-time events. These traces are then processed in a post-mortem analysis. As such, these approaches are neither dynamic nor adaptable to changes in the analyzed applications.

Live Feature Analysis is an approach that overcomes these drawbacks by adapting the application at run-time. Instead of generating traces, feature information is directly added to the structural representation of the source code while the features

are exercised. Live Feature Analysis has been originally implemented with Reflectivity [Denker *et al.*, 2010]. Two drawbacks of the previous approach are that it is difficult to specify an object-specific adaptation and that the management of links in Reflectivity is by hand and error-prone since there is no reification that models and adaptation composed of multiple links. Using the meta-object abstraction of Bifröst, the meta-level management can be handled by the meta-object itself on a per-object basis.

The following listing shows the next scenario: we instrument a given object before the execution of a feature with the goal of annotating the AST nodes of the code fragments that are executed by that feature:

```
1 aMetaObject := BehavioralMetaObject new.
2 aMetaObject
3    when: ASTNodeExecutionEvent new
4    do: [ :node | node addFeatureAnnotation: #printing ].
5 aMetaObject bindTo: anObject.
```

Listing 4.9: Live Feature Analysis

To implement live feature analysis we define a new behavioral object (Line 1) which is triggered every time an AST node is executed (Lines 2–4). The meta-level behavior is specified using a block closure (anonymous function). In this example, the feature 'printing' is added to the execute node (Line 4). Finally, the meta-object is bound to a particular object, `anObject`, to be analyzed (Line 5).

The `ASTNodeExecutionEvent` is an event triggered for every execution of an AST node, and it contains the knowledge of which are the AST nodes that reify this particular event. The behavioral meta-object then binds these nodes to an AST meta-object that will perform the meta-action described in the block. The block uses a helper method `addFeatureAnnotation:` which simply adds a symbol in the properties of the node. The code is automatically installed by Bifröst using an AST transformation. The block `node` parameter is a dynamic reification of the executed AST node created at runtime. Other reifications like the dynamic execution context are available too.

This example shows how Bifröst supports partial reflection by adapting a single object. The AST node is reified for each occurrence of the newly defined event thus we selectively reify. The adaptation is achieved without anticipation and at runtime.

4.2.6 Scoped Live Feature Analysis (Scenario 2)

A key drawback of live feature analysis is that the user still needs to specify where this adaptation should take place before exercising the features.

Prisma aids the user when the target of the feature analysis is unknown. We need to define the same meta-object used by live feature analysis inside a Prisma execution. However, the portions of the system that should be adapted are not selected by the user but instead by the execution itself. We use the term "AST execution" figuratively, since AST nodes are not literally executed, but rather their lower level bytecode representation is. However, when we adapt an application we specify that we would like something to happen when the bytecode, *i.e.*, the result of compiling a particular AST node, is executed.

```
loginExecution := Execution new.
loginExecution when: ASTNodeExecutionEvent
   do: [ :node | node addFeatureAnnotation: #login ]
```

The `Execution>>when: anEvent do: aBlock` method is responsible for adding a meta-object to the execution which should evaluate the provided block when a particular meta-event is produced. Whenever an AST node is executed for an adapted object the meta-level behavior is executed.

```
loginExecution
   executeOn: [ WebServer new loginAs: 'admin' password: 'pass' ]
```

Listing 4.10: Exercising the login feature on a web server.

Prisma applies this meta-object only to the specific method invoked during the execution. The meta-objects associated to the execution are never applied to a complete object unless the meta-object specifies so. In Listing 4.10 we are exercising the login feature on a web server. We are dynamically scoping the adaptation in the execution to the behavior in the block.

We present a more in-depth explanation of Prisma in Chapter 9.

4.3 Implementation

Bifröst offers a form of partial behavioral reflection that supports a highly selective means to specify where and when to reify system constructs. Bifröst generalizes and simplifies Reflectivity by using meta-objects (rather than links) as the sole abstraction with the responsibility of specifying the structure and behavior of any other object. Bifröst builds on an earlier prototype, called Albedo [Ressia *et al.*, 2010]. The main differences between these two prototypes are that Bifröst provides (i) a meta-object composition mechanism, (ii) a unification of the various meta-objects in the system, and (iii) a purely object-centric approach for applying meta-objects to objects.

4.3.1 Adapting the Lower-level

Bifröst's adaptation mechanism is built on top of lower-level meta-objects. In the Smalltalk implementation of Bifröst we bind meta-objects to objects representing AST nodes. AST nodes are static representations of the running behavior of a method in a class. Bifröst achieves object-centric reflection by duplicating the AST behavior representation in the meta-object and modifying it. Objects thereby become owners of their behavior instead of depending on a meta-class model.

A meta-object can be associated to a single AST node or multiple ones (see Figure 4.2). The next time the method is compiled the system automatically generates new bytecodes that take the meta-object into consideration. This behavior allows Bifröst to adapt the predefined behavior of objects. AST meta-objects can reify AST related information depending on the AST node. For example, a message send node can reify the sender, the receiver and the arguments at runtime. The meta-level behavior specified in the meta-object can be executed before, after or instead of the AST node the meta-object is adapting.

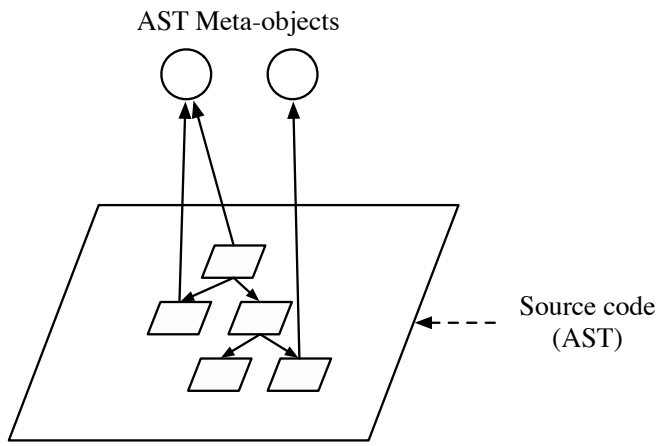

Figure 4.2: Bifröst AST adaptation through meta-objects.

The following section explains how this dynamic recompilation works in the context of Smalltalk.

4.3.2 Reflective methods

Bifröst exploits the *reflective method* abstraction [Marschall, 2006]. A reflective method knows the AST of the method it represents (see Figure 4.3). In Pharo classes are first class objects that are accessible and changeable at run-time. Classes hold a reference to a `MethodDictionary`, a special subclass of Dictionary. All methods of a class are stored in its method dictionary. The VM directly accesses class objects and method

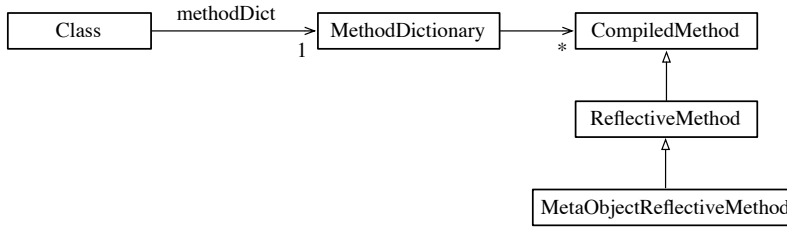

Figure 4.3: Reflective Methods in Method Dictionaries.

dictionaries when evaluating message sends. Normally, only instances of `CompiledMethod` are stored in the method dictionary of a class but Pharo allows us to replace it with any other object that obeys the right protocol. When such an object is used in place of a regular compiled method, the VM sends it the message `run:with:in:`, encoding the message, its arguments and the recipient. When a reflective method receives this message it processes the adaptations specified by the meta-object on the AST and generates a new compiled method that is eventually executed. If no adaptation is present the reflective method caches the compiled method to improve performance.

4.3.3 Structural and Behavioral Reflection

Behavioral reflective changes are achieved by attaching meta-objects to AST nodes, thus modifying or adding behavior to the target object's method.

On the other hand, structural reflection is handled differently. For example, when a method is added to a particular object then the meta-object is responsible for managing this method. The class is modified to understand the method and a reflective method is installed in the method dictionary. This reflective method delegates to the meta-object of the receiver object the responsibility of finding the method to be executed. If the meta-object has a method with the correct selector then the associated compiled method is executed. Otherwise a *does not understand* error is triggered, which is the original behavior. All instances of the class that were not adapted return *does not understand* when the added method is invoked.

In the case of adding an instance variable to a particular object, once again the meta-object is responsible for holding this variable. Behavioral adaptations are introduced in the methods that access this instance variable to delegate the access to the instance variable in the object's meta-object.

4.3.4 Object-specific Behavior

A meta-object defines how an object is interpreted. The meta-object abstraction has a method dictionary in which the corresponding reflective method for that specific object is stored. In Figure 4.3 we can see a reflective method abstraction called `MetaObjectReflectiveMethod`. When an object method is adapted the reflective method in the method dictionary is replaced by a `MetaObjectReflectiveMethod`. A copy of the reflective method is installed in the method dictionary of the object's meta-object. Finally the adaptation is performed over the copied AST method node. The key responsibility of a `MetaObjectReflectiveMethod` is to delegate the method execution to the object's meta-object.

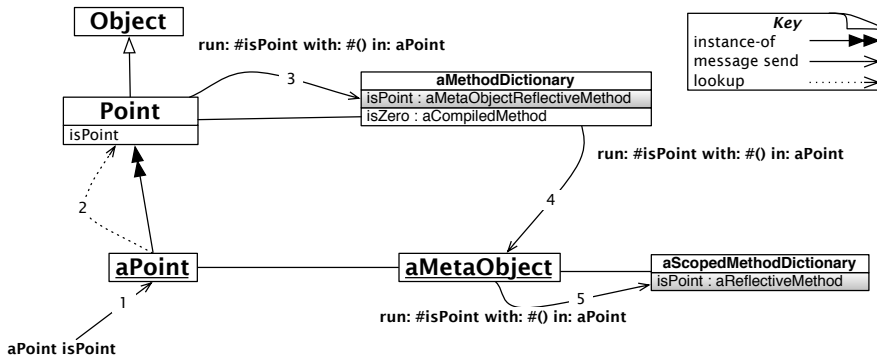

Figure 4.4: Modified method lookup for a point with an adapted `isPoint` method.

In Figure 4.4 we can see an example of the modified method lookup for `aPoint>>isPoint` in Bifröst. First the method lookup finds the method `isPoint` defined in the `Point` class. This method is not a compiled method but a reflective method. The VM does not know how to execute this abstraction thus it delegates the execution to the reflective method itself with `run:with:in:`. We can observe that the `MetaObjectReflectiveMethod` instance delegates to a meta-object through the message `run:with:in:`. In step 4 the reflective method delegates the execution of the method `isPoint` to the receiver's meta-object. To find the corresponding method to be executed the meta-object indexes by method name. The meta-object finds the corresponding method which is a reflective method containing the a copy of the original AST plus adaptations. The message `run:with:in:` is sent to the reflective method which first triggers the compilation of the method, second replaces the reflective method in the method dictionary with the resulting compiled method, and finally executes the compiled method.

If the message is sent to another object of the same class, for which no adaptation has been performed, the `MetaObjectReflectiveMethod` placed in the class method dictionary sends the message `run:with:in:` to the original reflective method cached in the reflective method.

There are other implementation mechanisms that we could have used to obtain the same behavior. Next we introduce these options and we explain why we did not use them:

- *Anonymous Classes* are a Smalltalk mechanism for changing the class of an object at runtime. When changing the class of an object the original class shape and the new class shape should be the same. This implies that the number of instance variables defined in the class cannot change. Also managing anonymous classes is not trivial, the IDE does not take into account these classes. Moreover, changing an object's class has important consequences over a class-based language, for example the inheritance chain is severed disrupting the normal behavior of the object when delegating to inherited behavior.
- *Lookup Method modification* is another option for achieving dynamic meta-objects. Instead of using reflective methods we can redirect the method lookup to an object's meta-object before reaching the object's class. However, this implies changing the VM since the method lookup in Smalltalk is not reified. We wanted to have a system that can be loaded and used in any Smalltalk environment regardless of the VM they are using.
- *Proxies* are objects that are placed before other objects. For example, if we want to count the number of times a specific object is invoked we create a proxy and change all references to the specific object to point to the proxy. This means that to invoke a method on the specific object we always have to go through the proxy which counts the invocations. In Smalltalk, there is a mechanism known as *become* which transforms all references to an object to another object, which in this case is a proxy. The problem with become is that it is slow and there is no way of doing it lazily. Due to this, we decided to use the reflective method mechanism which has no performance impact if the method is not used and when used the performance impact is smaller than changing all the references to an object.

4.3.5 Micro-Benchmark

We have performed a micro-benchmark to assess the maximal performance impact of Bifröst. We follow the test setup of Tanter [Tanter *et al.*, 2003] and base our benchmarks on the *message send* reification only, the other reifications having similar performance characteristics. All benchmarks were performed on an Apple MacBook Pro, 2.16 GHz Intel Core Duo in Pharo 1.1.1 with the jitted Cog VM. To avoid possible execution artifacts disturbing the benchmark we ensure that the involved reflective and jitted methods are created in advance and that method lookup caches are filled.

In our benchmark we measure the execution time of a test method being invoked one million times from within a loop. This test method is performing a simple constant time arithmetic operation to avoid the VM optimizing the method call. In Table 4.2

we report the average μ and standard deviation τ of running this benchmark one hundred times with three different setups:

	μ	τ
1. No reification	21.54	1.05
2. Disabled reification	21.53	1.07
3. Message send reification	729.60	16.52

Table 4.2: μ is the average time in milliseconds and τ is the standard deviation for 10^6 activations of the test method over 100 runs.

1. In the first test case (*no reification*) we measure the execution time of the application without Bifröst.
2. In the second test case (*disabled reification*) we measure the execution time of the application with Bifröst, but without reification on our benchmarked method. We see that there is no performance impact on parts of the system that do not use reflection.
3. In the third test case (*message send reification*) we measure the execution time of the application with Bifröst reifying the 10^6 method activations of the test method. This shows that in the reflective case the code runs about 35 times slower than in the reified one.

This micro-benchmark shows that reflection on a runtime system can have a significant performance impact. However, as we have demonstrated in the past and we will exemplify in the next section, the performance impact for real-world application with fewer reifications is lower and in some cases imperceptible for the user. Bifröst's meta-objects provide a way of adapting selected objects thus allowing reflection to be applied within a fine-grained scope only. This provides a natural way of controlling the performance impact of reflective changes.

4.3.6 Bifröst for other languages

In this section we discuss the feasibility of implementing Bifröst in other languages and environments.

There has been extensive work in AOP to support adaptation mechanisms. As Tanter [Tanter, 2008] has pointed out, there are several techniques to support dynamic deployment of aspects: residues [Masuhara *et al.*, 2003], meta-level wrappers [Hirschfeld, 2003], optimized compilers with static analysis [Avgustinov *et al.*, 2005; Bodden *et al.*, 2007], and VM support [Bockisch *et al.*, 2004]. Moreover, there has been promising work on aspect-aware VMs [Bockisch *et al.*, 2006b; Bockisch *et al.*, 2006a] and dynamic layer (de)activation [Costanza *et al.*, 2006], suggesting that such advanced scoping mechanisms can be efficiently supported.

Object-centric reflection is not hard to achieve in other languages. The key problem of this approach for other languages is its requirement for unanticipated changes. A more static mainstream language (*e.g.*, Java) solution would likely be more static in nature. There are numerous instrumentation libraries for Java bytecode, such as BCEL, ASM, or Javassist [Chiba, 2000]. The key problem in this approaches is that adaptations can be only introduced at either build-time or at load-time. Achieving the same dynamic behavior and unanticipation as in Bifröst's Smalltalk implementation is not possible.

4.4 Conclusion

In this chapter we have presented Bifröst the Smalltalk implementation of object-centric debugging. We explained the particularities of the Smalltalk implementation. We validated the object-centric approach through a set of examples that show that: (i) different reflection models are achievable, (ii) all the reflection requirements are covered by Bifröst, (iii) object-centric reflection allows developer to reflect on objects and their static representation (also objects) with a unified model, and (iv) the absence of the object paradox in the presented examples (the validation and demonstration of how the object paradox is avoided is presented in Chapter 8 and Chapter 7).

In the next chapters we discuss structural adaptations (Chapter 5) and behavioral adaptations (Chapter 6). In these chapters we concentrate on how the object paradox is implicitly present in the language abstractions and how object-centric reflection can avoid the paradox while providing an improved development approach.

Chapter 5

Dynamically Composable Units of Reuse

In this chapter we demonstrate how the object paradox is present in the reuse of behavior and state. Generally class-based solutions are used as the main target of reuse techniques, like traits. We demonstrate how object-centric reflection improves the reuse mechanism by providing a dynamic approach which avoids the object paradox.

Classes in object-oriented languages define the behavior of their instances. Inheritance is the principle mechanism for sharing common features between classes. Single inheritance is not expressive enough to model common features shared by classes in a complex hierarchy. Several forms of multiple inheritance have consequently been proposed [Borning and Ingalls, 1982; Keene, 1989; Meyer, 1997; Schaffert *et al.*, 1986; Stroustrup, 1986]. However, multiple inheritance introduces problems that are difficult to resolve [Dixon *et al.*, 1989; Sweeney and Gil, 1999]. One can argue that these problems arise due to the conflict between the two separate roles of a class, namely that of serving as a factory for instances, as well as serving as a repository for shared behavior for all instances and the instances of its subclasses. As a consequence, finer-grained reuse mechanisms, such as flavors [Moon, 1986] and mixins [Bracha and Cook, 1990], were introduced to compose classes from various features.

Although mixins succeed in offering a separate mechanism for reuse they must be composed linearly, thus introducing new difficulties in resolving conflicts at composition time. Traits [Schärli *et al.*, 2003; Ducasse *et al.*, 2006b] overcome some of these limitations by eliminating the need for linear ordering. Instead dedicated operators are used to resolve conflicts. Nevertheless, both mixins and traits are inherently static, since they can only be used to define new classes and not to adapt existing objects.

Ruby [Matsumoto, 2001] relaxes this limitation by allowing mixins to be applied to individual objects. Object-specific mixins however still suffer from the same compositional limitations of class-based mixins, since they must still be applied linearly to resolve conflicts.

We introduce *talents*, object-specific units of reuse that model features an object can acquire at run-time. Similar to traits, a talent represents a set of methods that constitute part of the behavior of an object. Unlike traits, talents can be acquired (or lost) dynamically. When a talent is applied to an object, no other instance of the object's class are affected. Talents may be composed of other talents, however, as with traits, the composition order is irrelevant. Conflicts must be explicitly resolved.

Like traits, talents can be flattened, either by incorporating the talent into an existing class, or by introducing a new class with the new methods. However, flattening is purely static and results in the loss of the dynamic description of the talent on the object. Flattening is not mandatory, on the contrary, it is just a convenience feature which shows how traits are a subset of talents.

The remainder of this chapter is structured as follows: In Section 5.1 we motivate the problem. Section 5.2 explains the talent approach, its composition operations and a solution to the motivating problem. In Section 5.3 we present the internal implementation of our solution in the context of Smalltalk. In Section 5.4 we discuss related work. Section 5.5 discusses about features of talents such as scoping and flattening. In Section 5.6 we present examples to illustrate the various uses of talents. Section 5.7 presents a dedicated user interface for managing and defining talents. Section 5.8 summarizes the chapter and discusses future work.

5.1 Motivating Examples

In this section we analyze two examples that demonstrate the need for a dynamic reuse mechanism.

Moose is a platform for software and data analysis that provides facilities to model, query, visualize and interact with data [Nierstrasz *et al.*, 2005; Gîrba, 2010]. Moose represents source code in a model described by FAMIX, a language-independent meta-model [Tichelaar *et al.*, 2000]. The model of a given software system consists of entities representing various software artifacts such as methods (through instances of `FAMIXMethod`) or classes (through instances of `FAMIXClass`). Each type of entity offers a set of dedicated analysis actions. For example, a `FAMIXClass` offers the possibility of visualizing its internal structure, and a `FAMIXMethod` offers the ability to browse its source code. Selecting the needed features for an entity is awkward within the constraints of a fixed class hierarchy.

In a second example, we consider various kinds of streams, whose features can be combined at run time, rather than requiring that a class be created for every conceivable combination of features.

5.1.1 Moose Meta-model

Moose can model applications written in different programming languages, including Smalltalk, Java, and C++. These models are built with the language independent FAMIX meta-model. However, each language has its own particularities which are introduced as methods in the different entities of the meta-model. There are different extensions which model these particularities for each language. For example, the Java extension adds the method `isSessionBean` to the `FAMIXClass`, while the Smalltalk extension adds the method `isExtended`. Smalltalk however does not support namespaces, and Java does not support class extensions. Additionally, to identify test classes Java and Smalltalk require different implementations of the method `isTestClass` in `FAMIXClass`.

Another problem with the extensions for particular languages is that the user has to deal with classes that have far more methods than the model instances actually support. Dealing with unused code reduces developer productivity and it is error prone.

A possible solution is to create subclasses for each supported language. However, there are some situations in which the model requires a combination of extensions: Moose JEE [Perin, 2010; Perin *et al.*, 2010] — a Moose extension to analyze Java Enterprise Applications (JEAs) — requires a combination of Java and Enterprise Application specific extensions. This leads to an impractical explosion of the number of subclasses. Moreover, possible combinations are hard to predict in advance.

Multiple inheritance can be used to compose the different behaviors a particular Moose entity requires. However, this approach has been demonstrated to suffer from the "diamond problem" [Snyder, 1986; Bracha and Cook, 1990] (also known as "fork-join inheritance" [Sakkinen, 1989]), which occurs when a class inherits from the same base class via multiple paths. When common features are defined in different paths then conflicts arise. This problem makes it difficult to handle the situation where two languages to be analyzed require the addition of a method of the same name.

Mixins address the composition problem by applying a composition order, this however might lead to fragile code and subtle bugs. Traits offer a solution that is neutral to composition order, but traits neither solve the problem of the explosion in the number of classes to be defined, nor do they address the problem of dynamically selecting the behavior. Traits are composed statically into classes before instances can benefit from them.

We need a mechanism capable of dynamically composing various behaviors for different Moose entities. We should be able to add, remove, and change methods. This new Moose entity definition should not interfere with the behavior of other entities in other models used concurrently. We would like to be able to have coexisting models of different languages, formed by Moose entities with specialized behavior.

5.1.2 Streams

Streams are used to iterate over sequences of elements such as sequenced collections, files, and network streams. Streams offer a better way than collections to incrementally read and write a sequence of elements.

Streams may be either readable, writeable or both readable and writeable. They can also be binary or character-based. Furthermore, streams can have different backends, such as memory streams, socket streams, database streams, or file streams.

The potential combination of all these various types of streams leads to an explosion in the number of classes.

Similar solutions to the Moose meta-model problem can be provided, however they present the same shortcomings. Multiple inheritance can be used to compose the different behaviors of a particular stream. However, the diamond problem again makes it difficult to handle the situation where two streams want to add a method of the same name. Mixins address the composition problem by applying a composition order, this however might lead to fragile code and subtle bugs. Although inheritance works well for extending a class with a single orthogonal mixin, it does not work so well for composing a class from many mixins. The problem is that usually mixins do not quite fit together, *i.e.*,, their features may conflict, and inheritance is not expressive enough to resolve such conflicts.

Traits offer a solution that is neutral to composition order, but traits neither solve the problem of the explosion in the number of classes to be defined, nor do they address the problem of dynamically selecting the behavior. Traits are composed statically into classes before instances can benefit from them.

We need a mechanism capable of dynamically composing the right combination of streams required for each particular occasion. The key objective is to avoid an exponential increase in the number of classes needed to provide all the different combinations.

5.2 Talents in a Nutshell

In this section we present a new approach of composable units of behavior for objects, called *talents*. These abstractions solve the issues present in other approaches.

5.2.1 Defining Talents

A talent specifies a set of methods which may be added to, or removed from, the behavior of an object. We will illustrate the use of talents with the Moose extension example introduced in the previous section.

A talent is an object that specifies methods that can be added to an existing object. A talent can be assigned to any object in the system to add or remove behavior.

```
aTalent := Talent new.
aTalent
   defineMethod: #isTestClass
   do: '^ self inheritsFromClassNamed: #TestCase'.
aClass := FAMIXClass new.
aClass acquire: aTalent.
```

We can observe that first a generic talent is instantiated and then a method is defined. The method `isTestClass` is used to test if a class inherits from `TestCase`. In lines 5–6 we can see that a FAMIX class is instantiated acquiring the previous talent. When the method `acquire:` is called, the object — in this case the FAMIX class — is adapted. Only this `FAMIXClass` instance is affected, no other instance is modified by the talent. No adaptation will be triggered if an object tries to acquire the same talent several times.

Talents can also remove methods from the object that acquires them.

```
aTalent := Talent new.
aTalent excludeMethod: #duplicatingClasses.
aClass := FAMIXClass new.
aClass acquire: aTalent.
```

In this case the existing method `duplicatingClasses` is removed from this particular class instance. Sending this message will now trigger the standard `doesNotUnderstand:` error of Smalltalk.

5.2.2 Composing Objects from Talents

Talent composition order is irrelevant, so conflicting talent methods must be explicitly disambiguated. Contrary to traits, the talent definition of a method takes precedence if the object acquiring the talent already has the same method. This is because we want behavior that is specific to objects, and as such the object-specific behavior should take precedence over the statically defined one. Once an object is bound to a talent then it is clear that this object needs to specialize its behavior. This precedence can be overridden if it is explicitly stated during the composition by removing the definition of the methods from the talent.

In the next example we compose a group with two talents. One expresses the fact that a Java class is in a namespace, the other that a JEE class is a test class.

```
javaClassTalent := Talent new.
javaClassTalent
   defineMethod: #namespace
```

```
4    do: '^ self container'.
5 jeeClassTalent := Talent new.
6 jeeClassTalent
7    defineMethod: #isTestClass
8    do: '^ self methods anySatisfy: [ :each | each isTestMethod ]'.
9 aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

In line 10 we can observe that the composition of talents is achieved by sending the comma message (`,`). The composed talents will allow the FAMIX class instance to dynamically reuse the behavior expressed in both talents.

5.2.3 Conflict Resolution

A conflict arises if and only if two talents being composed provide different implementations for the same method. Conflicting talents cannot be composed, so the conflict has to be resolved to enable the composition.

To gain access to the different implementations of conflicting methods, talents support an alias operation. An alias makes a conflicting talent method available by using another name.

Talent composition also supports exclusion, which allows the user to avoid a conflict before it occurs. The composition clause allows the user to exclude methods from a talent when it is composed. This suppresses these methods and allows the composite entity to acquire the otherwise conflicting implementation provided by another talent.

Our goal is to define models originating from JEE applications to support both Java and JEE extensions. Composing these two talents however generates a conflict for the methods `isTestClass` for a FAMIX class entity. The next example produces a conflict on line 10 since both talents define a different implementation of the `isTestClass` method.

```
1 javaClassTalent := Talent new.
2 javaClassTalent
3    defineMethod: #isTestClass
4    do: '^ self methods anySatisfy: [ :m | m isAnnotatedWith: #Test ]'.
5 jeeClassTalent := Talent new.
6 jeeClassTalent
7    defineMethod: #isTestClass
8    do: '^ self inheritsFromClassNamed: #TestCase'.
9 aClass := FAMIXClass new.
10 aClass acquire: javaClassTalent , jeeClassTalent.
```

If an unresolved composition conflict arises an exception is thrown when the composition is attempted.

There are different ways to resolve this situation. The first is to define aliases, like in traits, to avoid the name collision. Aliases are used to avoid collisions, rather than to resolve collisions by, say, using a priority mechanism:

```
10 aClass acquire: javaClassTalent , (jeeClassTalent @ {#isTestClass ->
       #isJeeTestClass}).
```

When the talent is acquired the method `isJeeTestClass` is installed instead of `isTestClass`, thus avoiding the conflict. Any other method or another talent can then make use of this aliasing.

Another option is to remove those methods that do not make sense for the specific object being adapted. Using the following syntax the method is subtracted from the talent.

```
10 aClass acquire: javaClassTalent , (jeeClassTalent - #isTestClass).
```

By removing the definition of `isTestClass` from the JEE class talent the Java class talent method is correctly composed.

Each FAMIX extension can be defined as a set of talents, each for a single entity, *e.g.*, class, method, annotation, *etc.* For example, we have the Java class talent which models the methods required by the Java extension to FAMIX class entity. We also have a Smalltalk class talent as well as a JEE talent that model further extensions.

5.2.4 Stateful Talents

In the original traits model, state can only be accessed within stateless traits by accessors, which become required methods of the trait. As demonstrated by Bergel *et al.* [Bergel *et al.*, 2007], traits are artificially incomplete since classes that use such traits may contain significant amounts of boilerplate glue code. Talents also provide a mechanism for dynamically defining state which is similar to its static counterpart, stateful traits.

```
1 aTalent := Talent new.
2 aTalent defineState: #testClass.
3 aClass := FAMIXClass new.
4 aClass acquire: aTalent.
```

We can observe that first a generic talent is instantiated and then a state called `testClass` is defined. This instance variable is a boolean attribute/field used to test if a class is a test case. In lines 4–5 we can see that a FAMIX class is instantiated acquiring the previous talent. As with behavioral talents when the method `acquire:` is

called, the object — in this case the FAMIX class — is adapted. Only this `FAMIXClass` instance is affected, no other instance is modified by the talent. No adaptation will be triggered if an object tries to acquire the same talent several times.

Since this state is introduced on a live object, we also provide a mechanism for managing the initialization. When no default value is provided then the new talent-defined state is set to `nil`, the usual default value for uninitialized attributes in regular Smalltalk code. The developer can use the state definition with default behavior to control state initialization values.

```
1 aTalent := Talent new.
2 aTalent
3    defineState: #testClass
4    default: true.
5 aClass := FAMIXClass new.
6 aClass acquire: aTalent.
```

The method `defineState:default:` adds a state definition to a talent which when acquired by an object will have a default value. In the example the state `testClass` has the value `true` by default. To avoid sharing the default values between objects the message `defineState: aSymbol subjectTo: aBlock` allows the user to provide a block which will dynamically define the value of the state.

Methods defined afterwards for the talent can refer to the newly created state without the need for accessors.

```
1 aTalent := Talent new.
2 aTalent defineState: #testClass.
3 aTalent
4    defineMethod: #isTestClass
5    do: '^ testClass'.
6 aClass := FAMIXClass new.
7 aClass acquire: aTalent.
```

In lines 3–5 we can see the definition of the method `isTestClass` which returns the boolean value `testClass` state. No definition of accessors is required and talents can define methods directly accessing the state.

The user can define state accessors by various state helper methods:

```
1 aTalent := Talent new.
2 aTalent defineStateWithAccessors: #testClass.
3 aClass := FAMIXClass new.
4 aClass acquire: aTalent.
```

By using `defineStateWithAccessors:` the talent definition also adds the two accessors for reading and writing on the `testClass` state. The user can also use `defineStateWithReadAccessor:` and `defineStateWithWriteAccessor:` which are self-explanatory.

5.3 Implementation

Talents are built on top of the Bifröst reflection framework [Ressia *et al.*, 2010]. Each talent is modeled with a structural meta-object. In Talents, we particularly use partially reflection on specific objects, talents are applied without anticipation, and they are composed dynamically.

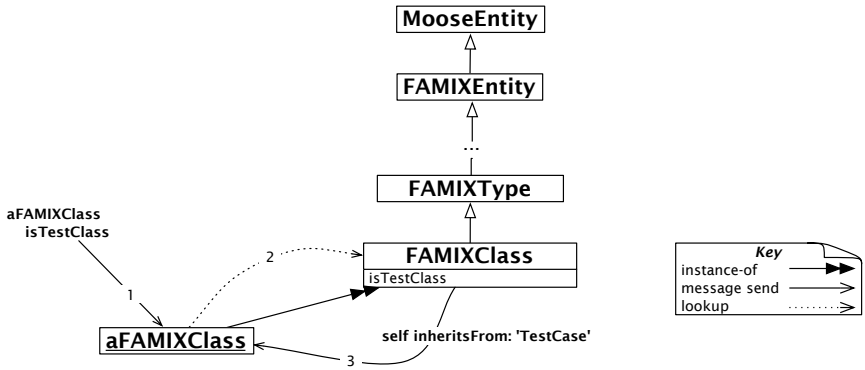

Figure 5.1: Default message send and method look up resolution.

Figure 5.1 shows the normal message send of `isTestClass` to an instance of `FAMIXClass`. The method lookup starts on the class defining the method and then executing it for the message receiver.

However, if we would like to factor the `FAMIXClass` JEE behavior out we can define a talent that models this. Each talent is modeled with a structural meta-object. A structural meta-object abstraction provides the means to define meta-objects like classes and prototypes. New structural abstractions can be defined to fulfill some specific requirement. These meta-object responsibilities are: adding and removing methods, and adding and removing state to an object. A composed meta-object is used to model composed talents. The specific behavior for defining and removing methods is delegated to the addition and removal of behavior in the structural meta-object.

In Figure 5.2 we can observe the object diagram for a FAMIX class that has acquired a talent that models JEE behavior. The method lookup starts in the class of the receiver. Originally, the `FAMIXClass` object did not define a method `isTestClass`, however, the application of the talent defined this method. This method is responsible for delegating the execution of the message to the receiver's talent. If the object does not have a talent, the normal method lookup is executed, thus talents do not affect other instances' behavior of the class. In this case, `aFAMIXClass` has a talent that defines the method `isTestClass`, which is executed for the message receiver.

Bifröst's structural meta-objects provide features for adding state to a single object and removing it. Talents can provide something that traits cannot, namely state.

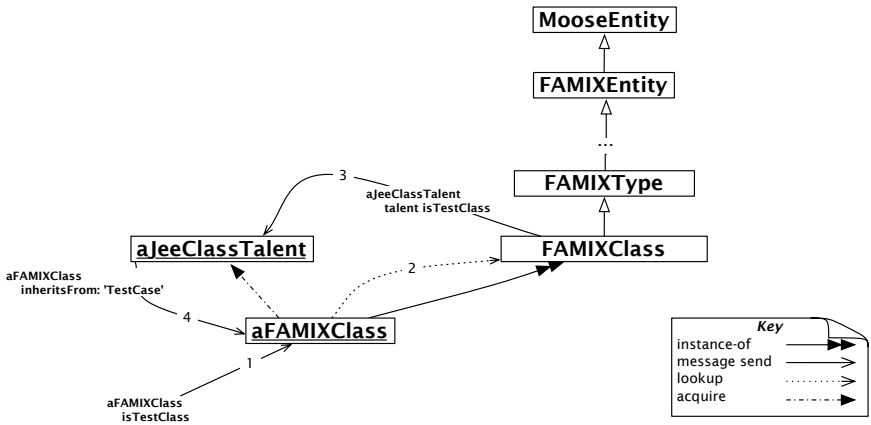

Figure 5.2: Talent modeling the Moose FAMIX class behavior for the method `isTestClass`.

Moreover, talents can provide operators for composing state adaptations. This composition is not present in object-specific techniques like mixins and Newspeak [Bracha *et al.*, 2010] modules.

5.4 Related Work

In this section we compare our approach using talents to other approaches for sharing behavior.

Mixins

Flavors [Moon, 1986] was the first attempt to address the problem of reuse across a class hierarchy. Flavors are small, incomplete implementations of classes, that can be "mixed in" at arbitrary places in the class hierarchy. More sophisticated notions of mixins were subsequently developed by Bracha and Cook [Bracha and Cook, 1990], Mens and van Limberghen [Mens and van Limberghen, 1996], Flatt, Krishnamurthi and Felleisen [Flatt *et al.*, 1998], and Ancona, Lagorio and Zucca [Ancona *et al.*, 2000].

Mixins present drawbacks when dealing with composition. Mixins use single inheritance for composing features and extending classes. Inheritance requires that mixins be composed linearly; this severely restricts one's ability to specify the glue code that is necessary to adapt the mixins so that they fit together [Schärli *et al.*, 2003]. However, although this inheritance operator is well-suited for deriving new classes from existing ones, it is not appropriate for composing reusable building blocks.

Bracha developed Jigsaw [Bracha, 1992], a modularity framework which defines module composition operators merge, override, copy as and restrict. These operators inspired the sum, override, alias and exclusion operators on traits. Jigsaw models a complete framework for module manipulation providing namespaces, declared types and requirements, full renaming, and semantically meaningful nesting.

Ruby [Matsumoto, 2001] introduced mixins as a building block of reusability, called modules. Moreover, modules can be applied to specific objects without modifying other instances of the class. However, object-specific modules suffer from the same composition limitation as modules applied to classes: they have to be applied linearly. Aliasing of methods is possible for avoiding name collisions, as well as removing methods in the target object. However, object or class methods cannot be removed if they are not already implemented. This follows the concept of linearization of mixins. Talents can be applied without an order. Moreover, a talent composition delivers a new talent that can be reused and applied to other objects. Filters in Ruby provide a mechanism for composing behavior into preexisting methods. However, they do not provide support for defining how methods defined in modules should be composed for a single object.

CLOS

CLOS [DeMichiel and Gabriel, 1987] is an object-oriented extension of Lisp. Multiple inheritance in CLOS [Lawless and Milner, 1989; Paepcke, 1993] imposes a linear order on the superclasses. This linearization often leads to unexpected behavior because it is not always clear how a complex multiple inheritance hierarchy should be linearized [Ducournau *et al.*, 1992]. CLOS also provides a mechanism for modifying the behavior of specific instances by changing the class of an instance using the generic function `change-class`. However, these modifications do not provide any composition mechanisms, rendering this technique dependent on custom code provided by the user.

Traits

Traits [Schärli *et al.*, 2003; Ducasse *et al.*, 2006b] overcome the limitations of previous approaches. A trait is a set of methods that can be reused by different classes. The main advantage of traits is that their composition does not depend on a linear ordering. Traits are composed using a set of operators — symmetric combination, exclusion, and aliasing — allowing a fair amount of composition flexibility. Traits are purely static since their semantics specify that traits can always be "flattened" to an equivalent class hierarchy without traits, but possibly with duplicated code. As a consequence traits can neither be added nor removed at run-time. Moreover, traits were not conceived to model object-specific behavior reuse.

Smith and Drossopoulou [Smith and Drossopoulou, 2005] proposed a mechanism for applying traits at runtime in the context of Java. However, only pre-defined behavior defined in a trait can be added at runtime. It is not possible to define and add new behavior at runtime.

Bettini *et al.* [Bettini *et al.*, 2009] proposed a mechanism for flexible dynamic trait replacement where traits can be applied at runtime. However, this technique can only change existing behavior, not add new behavior.

Object Extensions

Self [Ungar and Smith, 1987] is a prototype-based language which follows the concepts introduced by Lieberman [Lieberman, 1986]. In Self there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations. Objects are derived from other objects by cloning and modification. Objects can have one or more prototypes, and any object can be the prototype of any other object. If the method for a message send is not found in the receiving object then it is delegated to the parent of that object. In addition, Self also has the notion of trait objects that serve as repositories for sharing behavior and state among multiple objects. One or more trait objects can be dynamically selected as the parent(s) of any object. Selector lookups unresolved in the child are passed to the parents; it is an error for a selector to be found in more than one parent. Self traits do not provide a mechanism to fine tune the method composition. Let us assume that two objects are dynamically defined as parents of an object. If both parent object define the same method there is not a simple way of managing the conflict.

Object extension [Ghelli, 2002; Di Gianantonio *et al.*, 1998] provides a mechanism for self-inflicted object changes. Since there is no template serving as the object's class, only the object's methods can access the newly introduced method or data members. Ghelli *et al.* [Ghelli, 2002] suggested a calculus in which conflicting changes cannot occur, by letting the same object assume different roles in different contexts.

Drossopoulou proposed Fickle [Drossopoulou *et al.*, 2001], a language for dynamic object re-classification. Re-classification changes at run-time the class membership of an object while retaining its identity. This approach proposes language features for object re-classification to extend an imperative, typed, class-based, object-oriented language. Even though objects may be re-classified across classes with different members, they will never attempt to access non-existing members.

Cohen and Gil introduced the concept of object evolution [Cohen and Gil, 2009]. This approach proposes three variants of evolution, relying on inheritance, mixins and shakeins [Rashid and Aksit, 2006]. The authors introduce the notion of evolvers, a mechanism for maintaining class invariants in the course of reclassification [Drossopoulou *et al.*, 2001]. This approach is oriented towards dynamic reuse in languages with types. Shakeins provide a type-free abstraction, however, there are no composition operators to aid the developer in solving more complex scenarios.

Bracha *et al.* [Bracha *et al.*, 2010] proposed a new implementation of nested classes for managing modularity in Newspeak. Newspeak is class-based language with virtual classes. Class references are dynamically determined at runtime; all names in Newspeak are method invocations thus all classes are virtual. Nested classes were first introduced in Beta [Madsen *et al.*, 1993]. In Newspeak Class declarations can be nested to an arbitrary depth. Since all references to names are treated as method invocations any object member declaration can be overridden. The references in an object to nested classes are going to be solved when these classes are late bound to the class definition in the active module the object is in. Talents model a similar abstraction to modules, for dynamically composing the behavior of objects. However, Newspeak modules do not provide composition operators similar to talents. Composed talents can remove, alias, or override method definitions. Removing method definitions is not a feature provided by Newspeak modules. In Newspeak composition would be done in the module or in the nested classes explicitly.

Context-oriented Programming

Context-oriented programming (COP) was introduced by Costanza *et al.* [Costanza and Hirschfeld, 2005]. The behavior of an object is split into layers that define the object's subjective behavior. Layers can be activated and deactivated to represent the actual contextual state. When a message is sent, the active context determines the behavior of the object receiving the message.

Subjective Programming

Subjective behavior is essential for applications that must adapt their behavior to changing circumstances. Many different solutions have been proposed in the past, based, for example, on perspectives [Smith and Ungar, 1996], roles [Kristensen, 1995], contextual layers [Costanza and Hirschfeld, 2005], and force trees [Darderes and Prieto, 2004]. Depending on the active context, an object might answer a message differently or provide a modified interface to its users. These approaches mainly concentrate on dynamically modifying an object's behavior, however, there is no support for behavior reuse between object as it exists in traits or mixins.

Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 1997b] modularizes cross cutting concerns. Join points define all locations in a program that can possibly trigger the execution of additional cross-cutting code (advice). Pointcuts define at run-time if an advice is executed. Both aspects and talents can add new methods to existing classes. Most implementations of AOP such as AspectJ [Kiczales *et al.*, 2001] support weaving code at more fine-grained join points such as field accesses, which

is not supported by talents. Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system.

Aspects are concerns that cannot be cleanly encapsulated in a generalized abstraction (i.e., object, method, mixin). This means that in contrast to talents, aspects are neither designed nor used to build dynamic abstraction and components from scratch, but rather to alter the performance or semantics of the components in systematic ways.

5.5 Discussion

In this section we discuss other benefits that talents bring to a programming language.

5.5.1 Scoping Talents

Scoping talents dynamically is of key importance because it allows us to reflect on which context the added features should be active in and also to control the extent of the system that is modified. An object might need to have certain features in one context while having other features in a different context. Let us analyze an example to understand the motivation for scoping talents.

A bank financial system is divided into two main layers: the domain and the persistency layer. The domain layer models the financial system requirements and features. The persistency layer deals with the requirements of persisting the domain abstraction in a relational database. When testing the domain behavior of this application we do not want to trigger database-related behavior. Normally, this is solved through mocking or dependency injection [Fowler, 2005]. However, these solutions are not simple to implement in large and legacy systems which are not fully understood, and where any change can bring undesired side effects. Scoped talents can solve this situation by defining a scope around the test cases. When the tests are executed the database access objects are modified by a talent which mocks the execution of database related actions. In a highly-available system which cannot be stopped, like a financial trading operation, scoped talents can help in actions like: auditing for the central financial authority, introducing lazy persistency for updating the database, logging. This is similar to the idea of modules in Newspeak.

COP solutions can provide an implementation solution to bringing talents to other languages. Lincke *et al.* [Lincke *et al.*, 2011] presented a mechanism for composing layers in ContextJS, a JavaScript COP implementation. Talents offer a form of COP based on object-specific meta-objects rather than layers.

5.5.2 Flattening

Flattening is the technique that folds into a class all the behavior that has been added to an object. There are two types of flattening in talents:

Flattening on the original class. Once an object has been composed with multiple talents it has a particular behavior. The developer can analyze this added behavior and from a modeling point of view realize that all instances of the object's class should have these changes. This kind of flattening applies the talent composition to the object's class.

Flattening on a new class. On the other hand the developer might realize that the new responsibilities of the object are relevant enough to be modeled with a separate abstraction. Thus a new class has to be created by cloning the composed object behavior. This new class will inherit from the previous object class. Deleted methods will be added with a `shouldNotCallMethod` exception to avoid inheriting the implementation.

5.5.3 Talents in a statically typed language

A highly dynamic construct such as talents is possible in a statically typed language, even if the language implementation ensures that the type interface remains consistent. This can only be achieved by disallowing changes to the signatures of existing methods. Talents can safely replace existing methods as long as they do not alter their signatures.

In a statically typed language like Java we could declare talents with the help of a marker interface `Talent<T>` [Bracha, 2004], where `T` is generic type variable specifying the interface of the talent. These interfaces can then be modeled with talents. Particular combinations of talents can deliver different combinations of an object's interfaces.

```
interface Talent<T> implements T {
   // marker interface for a talent with the interface T
}
```

Classes that want to support a specific talent need to implement the marker interface `Talent` that they parametrize with the interface of the talent `T`. This forces the class to provide a default implementation of all the methods in the interface of the talent. The example with `FAMIXClass` in Section 5.1.1 would look in Java as follows:

```
interface TestTalent {
   boolean isTestClass();
}

class FAMIXClass implements Talent<TestTalent> {
```

```
   boolean isTestTalent() {
      return false;
   }
   ...
}
```

Finally, a generic static helper method must be provided to let objects acquire talents:

```
<O extends Talent<T>, T> void acquire(O object, T talent)
```

The above signature ensures statically that object `O` can only acquire the talent `T`, it was marked with talent `T` (implements `T`) already at compile time.

The outlined approach would enable talents in Java without weakening the existing type system. We imagine that the talents themselves could be implemented using bytecode rewriting of the methods in classes that implement the marker interface and the state pattern outlined in Section 5.6.3.

As we demonstrated in this section, talents are possible even in the context of a statically typed language. The implementation however will be limited to the predeclared talent interfaces only.

5.5.4 Traits on Talents

Conceptually, talents are a generalization of traits. Traits can only be applied to a specific set of objects, classes. Talents can be applied to any object in the system, including classes.

Traits can be implemented on top of the talent infrastructure by having a talent `TraitTalent` with a modified method `basicNew`. This talent is applied to the class in which we would like to have traits. The modified `basicNew` method has the extra behavior of applying the set of composed traits for the given class. This set of composed traits can be contained in an added state to the class defined by the `TraitTalent`. Each trait is defined as a talent and added to the modified class. A trait can also be defined as a wrapper on a talent which adds traits related method.

5.6 Examples

In this section we present a number of example applications of talents. These examples are selected to exercise the various facets of the talents mechanism, and as such, act as validation of the expressiveness of our approach.

5.6.1 Mocking

Let us assume that we need to test a class that models a solvency analysis of the assets of a financial institution customer. The method we need to test is `SolvencyAnalysis>>isSolvent: aCustomer`. This method delegates to `SolvencyAnalysis>>assetsOf: aCustomer` which executes a complex calculation of the various assets and portfolios of the customer. We are only interested in isolating the behavior of `isSolvent:`, not in the complexities of `assetsOf:`

When testing such a use case we need to modify the assets of a particular customer by increasing or decreasing the financial instruments deposits. This implies that we need to interact and create objects that are unessential in relation to the objective of the test case. Introducing more objects in a test case increases the chances of making the test fail for other reasons than the test objective. Talents provide a mechanism to modify the behavior of particular objects to modify this interaction, providing an object-specific mocking mechanism.

Let us analyze a talent solution to this use case:

```
1 SolvencyAnalysisTest>>testIsSolvent
2    | aCustomer anAnalysis |
3    aCustomer := Customer named: 'test'.
4    anAnalysis := SolvencyAnalysis new.
5    anAnalysis method: #assetsOf: shouldReturn: 1.
6    self assert: (anAnalysis isSolvent: aCustomer).
7    anAnalysis method: #assetsOf: shouldReturn: -1.
8    self deny: (anAnalysis isSolvent: aCustomer).
```

We added the method `method:shouldReturn:` to the class `Object` which creates a talent with a method named as the first argument and with the body provided by the second argument. In lines 5 and 7 you can see the use of this behavior. If the method `assetsOf:` return a positive amount then the customer is solvent otherwise not.

Talents can ease the testing of monolithic legacy applications built in a manner that does not easily support mocking.

5.6.2 Compiler Internal Abstractions

In traditional compiler design, the compilation of source code is a multi-step process: lexical analysis (scanning) is followed by syntactic analysis (parsing), which is followed by semantic analysis. This is followed by code generation, which itself may be split into multiple optimization and generation steps. Traditionally each of these steps has its own representation to work on the code, *i.e.*, the lexical analysis uses tokens, the syntactic analysis uses an abstract syntax tree, the semantic analysis uses an intermediate representation, and so on.

The problem with this approach is that it brings a significant overhead of performance and memory. At each step a plethora of new nodes need to be instantiated, as each step accumulates new state and requires different behavior. However, old state of previous steps cannot be thrown away. At any time in the compilation chain the compiler needs to be able to navigate back to the state of the previous steps, for example to pinpoint errors in the source file or to query the lexical structure. Compiler designers can address this requirement in two possible ways, but neither of them is very attractive: Either they copy and accumulate state along the compilation chain, which is error prone and slow; or they keep references to the nodes of the previous step, which can be expensive if the paths have to be navigated often.

With talents we have an elegant solution to this problem. Imagine the scanner reads a variable assignment such as `a := 12`. This results in 3 tokens to be created: `a`, `:=`, and `12`. These tokens not only contain the value they represent, but also know the source file and location in that file. In the syntactic analysis a parser detects that these 3 tokens form an assignment, built from the variable `a`, the assignment operator `:=`, and the value `12`. With talents we let the assignment token acquire an `AssignmentNodeTalent` that has — besides the node specific behaivor — also additional state: an assignment always consists of a variable node and an expression node. In this particular case the token `a` acquires the `VariableNodeTalent` and the value `12` acquires the `ValueNodeTalent`. In the next step, the semantic analysis, the `a` is further refined with the type of variable it represents. In this particular example the compiler could for example let it acquire the `InstanceVariableTalent`.

With each step in the compilation chain new talents are attached. The talents not only introduce new node specific behavior, but also add new state. The added state allows the objects to reorganize themselves in new ways. While the tokens are organized in a sequence of tokens, the syntactic nodes form an abstract syntax tree, and the semantic analysis forms a graph of references.

The approach with talents avoids the drawbacks of existing solutions. The same objects are passed through the complete compilation chain. Each step augments the objects with new state and behavior relevant for this step. Unnecessary copying of state and navigation between long object chains is avoided.

5.6.3 State Pattern

The state pattern [Gamma *et al.*, 1995] models the different states a domain object might have. When this object needs to do something then it delegates the decision of what to do to its state. A class per object state is created with the required behavior. Sometimes, multiple instances of each state are created and sometimes a singleton pattern is used.

Instead of having a state abstract class and then concrete subclasses for each of the more specific states we could use talents. We will have a single state class and then create as many instances as we have states. We can model each specific state with a

different talent that is applied to the state's instances, thus avoiding the creation of multiple state specific classes.

The talents based solution is simpler than a traditional state pattern, since it avoids the additional redirection from the object to its state. The developer simply has an object whose behavior changes, still having the state specific behavior but one indirection is eliminated.

5.6.4 Streams

Streams can be writable, readable, or both; depending on what talents are added. `WriteStreamTalent` adds the methods for writing to a stream, *i.e.*, `nextPut:` and `nextPutAll:`. `ReadStreamTalent` adds the methods to read from a stream, *i.e.*, `next` and `next:`. Streams can be binary or textual. Talents add the necessary supported methods, *i.e.*, `BinaryReadStreamTalent` adds `nextInt32`; and `TextualWriteStreamTalent` adds the methods `cr` and `space`. Furthermore streams are typically implemented on top of different backends, *i.e.*, collections, sockets, or files. Again we use talents that provide the necessary primitives for the read and write talents to actually perform the desired tasks.

All possible combinations of read, write, or read-write; binary or textual; memory, sockets, or files are possible. No unnecessary methods outside the requested capabilities are present. Furthermore, the talent composition avoids additional dispatching cost. The resulting streams are as efficient as if all 18 combinations would have implemented manually.

Traditional stream implementations check in every method if the underlying stream is still opened. With talents we can avoid such cumbersome checks and dynamically acquire a `ClosedStreamTalent` when a stream is closed. This talent either removes all modifying stream methods, or alternatively replaces them with one that throws an exception. This approach not only simplifies the implementation, but it is also more efficient as unnecessary tests are avoided altogether.

5.6.5 Class Extensions

Class extensions are a means to add required behavior to classes that belong to other packages outside our control. For example, when we load Moose there are several methods that are added or modified, in core classes like `Collection` hierarchy, `Object` , *etc.* This mechanism allows Moose developers to extend the system with Moose specific additions, *e.g.*, utility methods like *asMooseGroup* have been added to the core class `Collection` so to transform any collection in a MooseGroup. The implementation of the extended method `asMooseGorup` is shown in the snippet below:

```
1 Collection>>asMooseGroup
2    ^ MooseGroup withAll: self
```

Class extensions are also largely used within Moose to add functionalities from newly added packages to core packages. For example, when the Moose extension to analyze relational databases [Aryani *et al.*, 2011] is loaded, new methods like `maps` are added to the element `FAMIXNamedEntity` in the Moose core to keep track of the relations between relational elements and source code entities. The implementation of the extended method `maps` is shown in the snippet below:

```
FAMIXNamedEntity>>maps
   <MSEProperty: #maps type: #FAMIXMapping opposite: #mapSource> <multivalued>  <
     derived>
   <MSEComment: 'Map relationship.'>

   ^self privateState attributeAt: #maps ifAbsentPut: [FMMultivalueLink on: self
     opposite: #mapSource:].
```

A key drawback of this approach is that extensions do not support the definition of state, but only behavior. Moose developers need to implement more complex models since they cannot add state to classes outside Moose packages. Talents can be used to address this issue. The extension mechanism can be improved to provide state facilities by applying talents to all live instances of a particular class when an extension is loaded. When a state extension is defined for a particular class a talent with state definition is used. When the packages with extensions are loaded all instances of the extended classes are gathered and the predefined stateful talents are applied to them. Moreover, some class instances can be left out of the talent adaptation by providing conditions on various criteria. For example, only instances not reachable from core classes should be adapted.

5.7 User Interface

The talents browser is responsible for organizing, managing and defining talents. This browser is built using Glamour [Gîrba, 2010; Bunge, 2009], an engine for scripting browsers.

In Figure 5.1 we can observe an instance of the talents browser for the FAMIX class case study. The browser is vertically divided into two panes, the upper navigation section and the lower source code section.

The navigation section is divided into three panes following traditional Smalltalk browsers. The first pane shows a list of talents packages. Packages group related talents together. In this example we can see two talents packages: `BouncingAtoms` and `Moose`. Once a package is selected the talents pane is populated with the talents belonging to that package. In this example the `Moose` package is composed of two talents: `JavaClassTalent` and `J2EEClassTalent`. When a talent is selected the Methods pane is populated with the talent's method definitions. The icons before the name

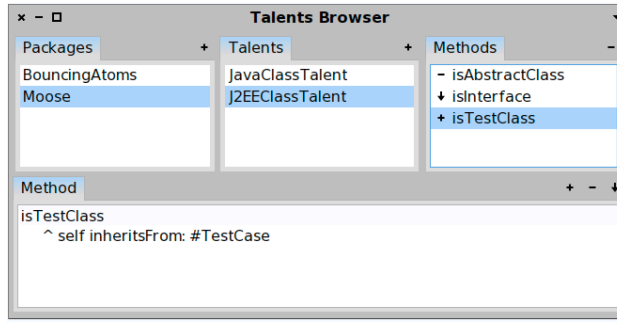

Figure 5.3: Talents Browser overview.

of the method represent the definition behavior. The — icon is used to signal that this method should be excluded when the talent is applied to an object. The □ icon indicates that a method should be replaced with the defined behavior. The + icon represents that the method should be added to the adapted object.

These panes provide contextual menus for removing, renaming and adding packages, talents and methods. The methods panel only provides a remove menu item. The talents panel provides a remove and a rename menu items.

The source code pane displays the source code defined for the selected method in the Methods pane. The three icons in the upper right corner represent the actions possible when saving a method definition. The + icon accepts the source code and adds a method definition to the selected talent. The — icon prompts the user to provide the name of the method which should be excluded by the selected talent. The □ icon accepts the source code and a method replacement definition to the selected talent.

Talents defined in the browser are registered to `TalentsRegistry`. When a developer needs to use a particular talent he can access the registry by name.

```
aTalent := TalentsRegistry registry talentNamed: 'J2EEClassTalent'.
```

The talents browser is useful for managing the creation and structure of talents but it does not provide a way to manage the association of talents to objects. To fulfill this we modified the default object inspector, one of the main instruments used during development, to open the talent browser directly on the inspected object. Figure 5.4 shows the work flow to attach a talent to an object. Once we have opened the system inspector on the object we want to enrich with a talent, we can open the talent browser directly from the contextual menu of the object. As a result the object we were inspecting is passed along to the browser. The talents browser allows the developer to find the right talents and if need be to modify it for fulfilling new requirements. The developer can then select a talent and associate it to the inspected

object using the option `Apply Talent` on the contextual menu that open on the talent classes.

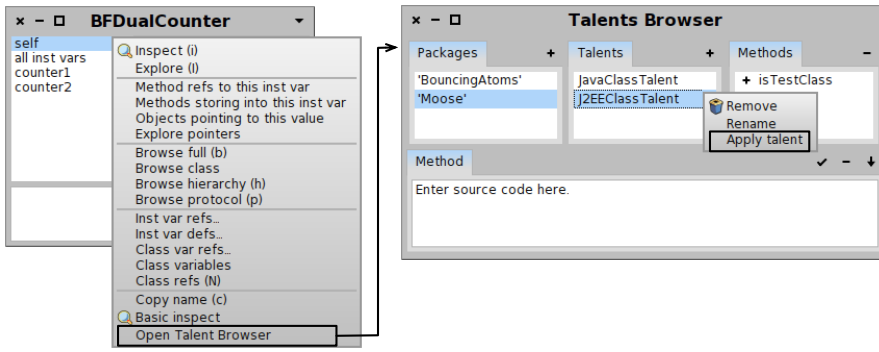

Figure 5.4: Modified inspector and Talents Browser Insteraction.

5.8 Conclusion

This chapter presented talents, a dynamic compositional model for reusing behavior. Talents avoid the object paradox since they exclusively target specific objects. Talents are composed using a set of operations: composition, exclusion and aliasing. These operations provide a flexible composition mechanism while avoiding the problems found in mixins and traits.

Talents are most useful in composing behavior for different extensions that have to be applied to the same base classes, thus dynamically adapting the behavior of the instances of these classes seems natural to obtaining a different protocol.

Managing talents can currently be complicated since the classic development tools are unaware of them. Our talents user interface solves the problem of managing and defining talents. However, it does not provide features for composing talents nor does it help in visualizing these compositions. We plan on extending the talents user interface to deal with composition requirements.

We plan on providing a more mature implementation of the talents scoping facilities. This technique shows great potential for the requirements of modern applications, such as dynamic adaptation and dependency injection for testing, database accesses, profiling, and so on.

Chapter 6

Decoupling Instrumentation from Software Analysis Tools

In this chapter demonstrate how object-centric reflection can be used to build a framework for addressing one of the applications of reflection: software analysis tools. When instrumenting a system the developer is limited in which events he can hook into by the abstractions provided by the reflective system. Generally these abstractions are related to events on classes, source code and particular instructions, even though the developer is interested in particular objects. The object-paradox is present in the instrumentation domain. Object-centric reflection provides the means for building a framework which decouples the instrumentation from the analysis tool itself with explicit object-specific meta-events.

Instrumentation is the process of adapting a software system to measure run-time attributes of interest. Development tools like profilers, loggers and code coverage analysis tools are traditional applications of instrumentation. Existing approaches to instrumentation are perfectly capable of implementing any of these use cases. Combining analyses, however, poses a number of difficulties.

Domain-Polluted Instrumentation. Existing event-based instrumentation approaches tend to couple the instrumentation behavior with the domain behavior. For example, a message counter profiler needs to instrument an application to reify message sends. If we would like to apply a message send logger to the same application we cannot reuse the already instrumented message send reification since it is coupled to the profiling domain. Due to this, a new instrumentation for the same reification is needed. Existing approaches pollute the instrumentation with domain behavior thus rendering this instrumentation only usable for a particular consumer, domain or context. Instrumentation behavior and its consumers are tightly coupled.

Language-biased Events. Event-based reflective approaches are coupled with specific characteristics of the host language. The language's internals define the number of canonical events and which abstractions they should represent. For example, some aspect-oriented programming (AOP) [Kiczales, 1996; Kiczales *et al.*, 1997b; Kiczales *et al.*, 1997a] languages depend on the structure and

organization of the code. The programmer must express concerns in terms of code-related events, usually method calls, rather than in terms of domain concepts. This problem is known as the *fragile pointcut problem* [Störzer and Koppen, 2004].

Static Instrumentation Scoping. Most instrumentation approaches use statically defined conditions to control the runtime impact of the instrumentation. Which portions of the system should trigger an event is defined through conditions which have to be manually maintained. There is no dynamic mechanism for plugging and unplugging instrumentations on specific objects.

We propose to resolve these problems by fully separating instrumentation from analysis with the help of explicit meta-level events. We simplify the meta-level's behavioral model by offering a single canonical event which models the execution of an abstract syntax tree (AST) node. Any other object-related event can be expressed in terms of this canonical event. Objects in an application are instrumented to reify meta-level events. Analysis tools select which events to observe for the purpose of profiling, logging, coverage, *etc.*

Our approach provides enhanced separation of concern capabilities by using runtime objects as the modularity unit. Instrumentation requirements are applied to specific objects thus allowing to reflect on any portion of the runtime system modeled with objects.

Outline. The remainder of this chapter is structured as follows: in Section 6.1 we discuss the related work with more details about their different implementations and problems. Section 6.2 shows the Chameleon approach in a nutshell. In Section 6.3 we show how Chameleon overcomes the drawback of other approaches. Section 6.4 presents how Chameleon is implemented. In Section 6.5 we summarize the chapter.

6.1 Related Work

In this section we review the state of the art in instrumentation techniques, while highlighting problems and open issues. In particular we shall see that existing approaches offer limited expressiveness in terms of the way that run-time events are made available for instrumentation purposes.

6.1.1 Applications of Instrumentation

Instrumentation has been extensibly researched in the past. One of the main concerns that researchers have focused on is the performance impact that instrumented code might have on program execution.

Removing instrumentation once it has fulfilled its purpose is key to reducing the performance impact. Dynamic instrumentation [Hollingsworth *et al.*, 1997; Stuart *et al.*, 2000] can be used to reduce the performance impact. Arnold and Ryder [Arnold and Ryder, 2001] have shown that combining instrumentation with sampling leads to accurate profiles (93–98% overlap with a perfect profile) with low overhead (3–6%).

DTrace [Cantrill *et al.*, 2004] is a tool capable of dynamically instrumenting base-level and kernel-level software. Tracing programs are defined in the D language, a subset of C with added functions and variables specific to tracing. DTrace users define probes which are instrumentation points. A probe is composed of a condition and an action. Probes are comparable to pointcuts in aspect-oriented programming. The DTrace framework itself performs no instrumentation of the system; that task is delegated to instrumentation providers. Providers are loadable kernel modules that communicate with the DTrace kernel module. Probes are advertised to consumers, who can enable them by specifying any element of a 4-tuple to scope the instrumentation: provider, module, function, name. The provider dynamically instruments the system and the probe's action is executed.

ATOM [Srivastava and Eustace, 2004] and Purify [Hastings and Joyce, 1992] instruments systems to collect data about them. Both these tools use static techniques, instrumentation happening when the analyzed system is not running.

6.1.2 Behavioral Reflection

Let us analyze a simple Iguana/J example proposed by Redmond and Cahill [Redmond and Cahill, 2002]. In this example a message execution event is reified. Every time that this event reification is triggered at runtime special verbose output after and before the method execution is shown.

```
1 class VerboseExecution extends MExecute {
2   Object execute(Object o, Object[] args, Method m)
3         throws InvocationTargetException,
4               IllegalAccessException {
5     System.out.println("Before method " + m.getName());
6     result = m.invoke(o, args);
7     System.out.println("After method " + m.getName());
8   }
9 }
```

Listing 6.1: Reification of method execution with Iguana/J

We can observe in this example that the problem domain solution is coupled with the event reification. Thus other potential consumers of this reification cannot reuse it. Moreover, they have to duplicate the same reification with their domain specific needs.

When scoping the event reifications, Iguana/J models Meta-object protocols (MOPs) that are associated to language constructs. We can see an example in Listing 6.2 where a `VerboseProtocol` is defined. It is composed of two event reifications: method execution and state write.

```
protocol VerboseProtocol {
  reify Execution: VerboseExecution;
  reify StateWrite: VerboseStateWrite;
}
```

Listing 6.2: Iguana/J MOPs definition.

In Listing 6.3 we observe how a protocol is associated with a single object. The Meta abstraction models the object responsible to managing the association between protocols and objects. Protocols can also be associated to classes, in which case they are applied to all instances of the class.

```
MyClass obj = new MyClass();
Meta.associate(obj, "VerboseProtocol");
```

Listing 6.3: Iguana/J Scoping.

A drawback of this approach is that the event consumer cannot scope dynamically which reifications he wants to listen to. This can be seen in Listing 6.2 of the Iguana/J example. The logger—actually not implemented but represented by the print function of the system—is bound to the Verbose Execution. There is no possibility provided to change the focus of the logger from the `VerboseExecution` to another event.

6.1.3 Aspect-oriented Programming

Aspect languages like AspectJ [Kiczales *et al.*, 2001], Composition Filters [Bergmans and Aksit, 2004] and CaesarJ [Aracic *et al.*, 2006] depend on the structure and organization of the code. The programmer must express concerns in terms of code-related events, usually method calls, that most of the times are too far away from their natural description. This problem is known as the *fragile pointcut problem* [Störzer and Koppen, 2004]. Let us use an example from Lieberherr *et al.* [Lieberherr *et al.*, 1999] for tracing particular method invocations on the class `Point`. The names of classes and operations that are affected are mentioned in the definition of the aspect.

```
1 aspect ShowAccess {
2    static before Point.get,
3                  Point.getX,
4                  Point.getY {
5      System.out.println("R");
6    }
```

```
7 }
```

Listing 6.4: Aspect logging the accesses to instance variables in the class Point

The aspect is only applicable to a single context, thus impairing the modularity of the aspect definition.

The JAsCo language [Suvée *et al.*, 2003] follows the work of Lieberherr *et al.* on the component view for separation of concerns. This language provides a way of separating *when* an aspect should be applied and *what* should be done. Hooks are defined with abstract pointcuts. Traps are introduced in the potential places where an event should be triggered. Connectors link these events to the hooks that dictate what should be done. Connectors can be loaded dynamically making this approach highly dynamic. It is not possible to define connectors at multiple abstraction layers.

Douence, Motelet and Südholt [Douence *et al.*, 2001] introduced a general operational model for crosscutting based on execution monitors called Event-based Aspect-Oriented Programming (EAOP). They proposed a formal model for the definition and detection of event patterns. They describe an event as the representation of a point in the program execution. In their prototype they implemented the explicit events method call and method return.

Douence and Südholt [Douence and Südholt, 2002] later introduced constructor calls and constructor returns as events. The Execution Monitor in their implementation observes events emitted during execution. The execution of the base program is suspended when an event is emitted. The monitor matches this event against different event patterns. When a pattern is satisfied the associated actions are executed.

The event is then propagated to all aspects. After each aspect in turn has reacted to the event the control is given back to the base program.

A key drawback of EAOP is that the event/pointcut definition is coupled to the consumer behavior thus the event abstraction is not reusable. This drawback is also present in some AOP approaches.

Another important drawback is that this approach provides only four events: constructor call and return, and method call and return. Although the authors claim that is possible to extend this set of events with state read and write they do not describe a solution to this. Neither do they provide a mechanism for developers to generate custom events. Moreover the reifications that are introduced in each of these events are fixed.

Bockisch *et al.* [Bockisch *et al.*, 2011] explicitly model events with information accumulation features and compose events and aspects into hierarchies to loosen the connection to code-level methods and field names. This approach proposes an explicit language construct for event declarations instead of just defining events using

declarative predicates, as pointcuts in AspectJ. Explicit events and aspects composition mechanism are provided to define more complex entities from simple ones.

Join point interfaces (JPI) [Inostroza *et al.*, 2011] address the strong contract between the pointcut definition and the base code. JPI provides an additional layer of abstraction between base code and aspects. A class can exhibit a particular join point interface. Any aspect can be defined referencing this interface thus the definition of the pointcut is moved to the base code class. The base code programmer can maintain pointcut definitions in sync with the base code that these pointcut definition refer to. The programmer is also aware of the pointcuts exposed to aspects. One shortcoming of this approach is that the developer has to predefine statically which is the possible interface of pointcuts that a class or object might have.

Gasiunas *et al.* [Gasiunas *et al.*, 2011] proposed EScala, a modular declarative model of implicit and imperative events. EScala extends the idea of events as object members, as realized by C# events, with the possibility to define events declaratively by expressions over other events. EScala presents the concept of declarative object-oriented events that complement imperative events with AO implicit events. EScala takes the AO way of defining implicit events but it does not solve the fragile pointcut problem.

We can summarize the key issues with existing approaches in three main points: domain-polluted instrumentation, language-biased events and static instrumentation scoping.

6.2 Chameleon in a Nutshell

Our goal is to provide an approach to instrumentation that addresses the three drawbacks of existing approaches. We propose to resolve these shortcomings by means of explicit meta-events. In this section we introduce Chameleon[1], a Smalltalk prototype of our approach.

Chameleon models meta-events explicitly and separates the specific behavior of a development tool from the instrumentation by applying the observer pattern. Chameleon provides a simplified event architecture with a single canonical event on top of which any other meta-level event can be defined. This is achieved by integrating two key approaches to reflection: CodA/Iguana's event-oriented approach and Reflex/Reflectivity's partial behavioral reflection approach.

Events are the building blocks of both CodA and Iguana, however, in neither framework are they modeled explicitly. By explicitly modeling events and applying the observer pattern, a better separation of concerns can be achieved.

Figure 6.1 shows a class diagram of Chameleon's key abstractions which we will discuss next.

1 `http://scg.unibe.ch/research/chameleon/`

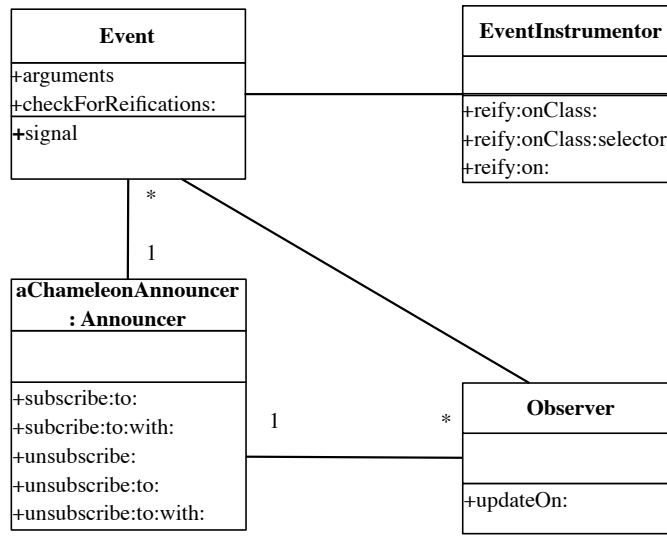

Figure 6.1: Chameleon's core abstractions.

6.2.1 Events

Chameleon offers a single canonical event which models the execution the code represented by single AST node. On top of the AST node execution event every other object-related event can be built. Also, by having explicit events, we can build developer tools to use the observer pattern to listen to these meta-events. Thus, a better separation of concerns between events generation and domain requirements is achieved.

The class of an event models the conceptual abstraction of that event. Each occurrence of an event is modeled with a new instance created from the event class. The responsibilities of an event class are:

- Determine where the event should be reified and signaled. The event class knows which AST nodes, when executed, should reify the event.
- Describe which dynamic data is required to reify the event. For example, a message send event contains and reifies the sender object, the receiver object and arguments of the message.
- Knowing what adaptation has to take place on an object so the event can be reified. For example, when reifying an object creation event it might happen that the message `new` is not overridden in the instrumented object. Because of this, the instrumentation tool should add the `new` method so the event can be reified.

Every time an event is triggered a new instance of this event is created. An event instance is responsible for knowing how to signal itself. Every event instance holds extra reifications which depend on the event, for example, sender of a method, variable written.

We have implemented the Iguana/J canonical events to show the capacities of Chameleon approach. These events are: object creation, object deletion, method execution, message send, message dispatch, state write and state read. For a detailed explanation of these events look at Section 6.4.3.

6.2.2 Instrumentation for Signaling Events

The `EventInstrumentor` is responsible for instrumenting an application to reify specific events under specific circumstances.

Instrumentation is performed at the AST level. The `EventInstrumentor` has the following responsibilities:

- Instrument AST nodes to reify an event. The instrumentation has two responsibilities: create the event and signal it through the announcer.
- Provide different scopes for instrumentation. For example, an event can be reified for several classes or a single class, or for a single method or for a single node.
- Adapt the application to allow an event to be reified. In the case of the `ObjectCreationEvent` the `new` method has to be added before the reification instrumentation could take place.

Chameleon provides different scopes when instrumenting an event. Each event can be reified either on a class, method or a single node. As an example we reify method execution on a bank account class.

```
(EventInstrumentor new) reify: MethodExecutionEvent onClass: BankAccount
```

Listing 6.5: Reification of method execution inside a class

The method `EventInstrumentor>>reify:onClass:` handles the instrumentation. Another possible scope is the method:

```
(EventInstrumentor new) reify: MessageSendEvent onClass: BankAccount
     selector:#statement
```

Listing 6.6: Reification of message send inside a method

In this example we reify the message send event on the statement method of the bank-account class.

The third scope can be defined for a single node:

```
(EventInstrumentor new) reify: ASTNodeExecutionEvent on: aNode
```

Listing 6.7: Reification of node execution on a single node

Here we reify the node execution on a node.

6.2.3 Announcer

The responsibility of the `Announcer` is to provide the observers with the possibility to subscribe and unsubscribe to events. The `Announcer` provides different scopes for the subscription. An observer can subscribe to all events or all occurrences of a particular event. In the listing below we can see a profiler subscribing to all events:

```
aChameleonAnnouncer
   subscribe: aProfiler
```

Listing 6.8: Profiler subscribing to all events.

For simplicity reasons a global instrumentor which uses a global announcer is defined. However, the design does not force the users to only use these global objects. Developers are free to instantiate new instrumentors and announcers creating contextual event refication environments.

Next, we can see a profiler subscribing to a method execution event.

```
aChameleonAnnouncer
   subscribe: aProfiler
   to: MethodExecutionEvent
```

Listing 6.9: Profiler subscribing from the method execution event.

To inform the announcer about an event execution, the initialization of an event has to call the `Announcer>>announce: anEvent` method.

An observing customer is able to unsubscribe from either all events or an event type as seen in the following listings:

```
aChameleonAnnouncer
   unsubscribe: aProfiler
```

Listing 6.10: Profiler unsubscribing to all events.

```
aChameleonAnnouncer
   unsubscribe: aProfiler
   to: MethodExecutionEvent
```

Listing 6.11: Profiler unsubscribing to the method execution event.

6.2.4 Observers

An observer models different development tools like profilers, code coverage analysis and loggers. Observers subscribe through the `Announcer` to listen to specific events. The particularities of the domain are kept in the observer thus providing good separation of concern between the instrumentation and the problem domain. The observers do not have to follow any particular pattern but the observer pattern to listen to events and act accordingly.

Observers are notified by the announcer when an event has been signaled and reified with the message `updateOn: anEvents`.

6.3 Chameleon in Action

In this section we will demonstrate how Chameleon resolves the three drawbacks of existing instrumentation approaches.

6.3.1 Domain-Polluted Instrumentation

MetaSpy [Bergel *et al.*, 2011] is a framework to build domain specific profilers. MetaSpy offers abstractions to model profilers and instrumentors. A profiler uses various instrumentors to reify domain specific events.

In a first case study of MetaSpy, the authors built a domain specific profiler for Mondrian [Meyer *et al.*, 2006], a software visualization tool. The default visualization displays a software system as nodes and edges representing classes and inheritance. Mondrian can be customized to change what the nodes and edges represent.

The authors of MetaSpy built a domain-specific profiler for Mondrian to detect the source of a certain performance issue. Each time Mondrian detects a change in a node it refreshes the whole visualization. This profiler's goal was to measure the number of times the `displayOn:` method was called for each of the nodes in a Mondrian visualization.

The MetaSpy domain-specific profiler was defined as follows:

```
1 MondrianProfiler>>setUp
2    self model root allNodes do: [ :node |
3       self
4          observeObject: node
5          selector: #displayOn:
6          do: [ :receiver :selector :arguments |
7             actualCounter
8                at: receiver
9                put: ((actualCounter
```

```
10                      at: receiver
11                      ifAbsent: [ 0 ]) + 1) ] ]
```

Listing 6.12: Attaching the profiler to the Mondrian-nodes

This profiler defines that every node in the model will be observed for invocations of the `displayOn:` method. The block from lines 6–11 defines the action that should be executed when the message `displayOn:` is received by any node. The block counts for each node how many times the method was executed.

The message `MetaSpy>>observeObject:selector:do:` delegated to an instrumentor which is responsible for instrumenting the nodes.

This process of instrumentation is described by the MetaSpy authors as follows:

> An instrumentation strategy is responsible for adapting a domain-specific model and triggering specific actions in the profiler when certain events occur.

This statement indicates coupling between the profiler and the instrumentation. The MetaSpy instrumentor introduces the profiler extra behavior in the nodes where the reification of interesting events should happen. This reification cannot be reused since it is coupled to this particular domain. We call this domain-polluted instrumentation, because the instrumentation is coupled with the profiler.

If we would add a second MetaSpy profiler the same event has to be reified again. This is due to the fact that each instrumentation hooks the functionality directly into the code. So every newly added profiler adds a new instrumentation and therefore additional code (besides the profiler) to the system. Therefore the coupling of instrumentation and the profiler has not only the drawback of not being able to reuse the instrumentation event reifications but it also has performance impact. Adding multiple profilers that use the same instrumentation adds the same reification for each profiler to the source code thus slowing the system unnecessarily. This performance impact is only noticeable when multiple adaptation are consuming the same events.

Chameleon on the other hand separates the instrumentation reification from the development tools. In the case-study of Mondrian we would need a method execution event to have the correct representation of the `displayOn:` method execution.

```
(Instrumentor new)
   reify: MethodExecutionEvent
   onClass: Node
   selector: #displayOn:
```

Listing 6.13: Reifying method execution for the method displayOn: in the class node of Mondrian

The Profiler only has to register for this event:

```
aChameleonAnnouncer
   subscribe: mondrianProfiler
   to: MethodExecutionEvent
```

Listing 6.14: Profiler registers for the method execution event.

If another development tool wants to observe this method execution, since we just have to register it, the instrumentation is already there. This is shown in the code below:

```
aChameleonAnnouncer
   subscribe: logger
   to: MethodExecutionEvent
```

Listing 6.15: Logger registers for the method execution event

The event does not need to be reified again. Therefore we avoid domain-polluted instrumentation.

The use of explicit events makes the reuse of instrumentation possible and provides the capability to always know what is already instrumented. It also reduces the performance impact whenever multiple development tools use the same instrumentation.

6.3.2 Language-biased Events

Lienhard *et al.* [Lienhard *et al.*, 2008] proposed an object-oriented back-in-time debugger. This kind of debugger is extremely useful for identifying the causes of bugs that corrupt execution state without immediately raising an error, as they allow us to inspect the past states of objects no longer present in the current execution stack. To remember the flow of objects this debugger has to answer a key question: How was this object passed here? This means that for any object accessible in the debugger, the tool has to be able to inspect all origins up until the allocation of the object. This also allows us to find out where a particular value of a variable comes from.

The approach of Lienhard *et al.* was to modify the Smalltalk virtual machine at key points to gather the state of variables and instance variable written and read. This also required to take into account the values of the arguments of method invocations.

Let us analyze the possibility of implementing a similar solution using Iguana/J. Message invocations as well as state read and write are provided as canonical events. State read and write events only model the accesses to instance variables but not regular variables. Specific behavior for gathering the required data can be introduced for these events. However, there is a key event that we cannot model with Iguana/J, when a value is assigned to a simple variable. It is not possible to reify this event

with the provided canonical events. Iguana/J canonical events are implemented by modifying the Java VM using the JIT interface. Thus, a variable assignment event would require one to modify the Java VM to introduce the new assignment event. In Iguana/J each of the seven canonical events matches a particular virtual machine adaptation. Moreover, some events like object creation are required since in Java an object is created through constructors instead of normal messages to the class.

Our approach proposes an unbiased event implementation by building on top of the AST abstractions. This means that events match AST nodes with particular characteristics. We can find AST abstractions in many different languages, therefore our framework can be ported to other languages.

To build the variable assignment event we need to extend `ASTNodeExecutionEvent` and override the method `shouldBeReifiedOn:`.

```
VariableAssignmentEvent>>shouldBeReifiedOn: aNode
   ^ aNode isAssignment
      and: [ aNode variable isVariable ]
```

Listing 6.16: Selection of assignment nodes for reifying the event that a variable was assigned.

Only the nodes that are assignments and whose variable side is a variable and not a field will be selected.

6.3.3 Static Instrumentation Scoping

Bockisch *et al.* proposed a new way to define events that can be composed and concatenated. The next snippet of code describes their motivating example of a shopping cart discount. Every time a product in low demand is purchased a discount is applied to the purchase value.

A `LowActivity` event is triggered when the product is in low demand. `LowActivityPurchase` is triggered after an `LowActivity` event. When a purchase is made and the product involved is in low demand then a `LowActivityPurchase` event is triggered. If these two events are sequentially triggered the `LowActivityDiscount` aspect is executed applying a discount.

```
1 event LowActivity(P product){
2   int LOWER_BOUND = 100;
3   Info purchaseInfo = new Info();
4   after(Purchase purchase): RelevantPurchase(purchase) {
5    purchaseInfo.increase(purchase.product());
6   }
7   when(P product): call(P.timeDone()) && target(product) {
8     if (purchaseInfo.count(product) < LOWER_BOUND) {
9       trigger(product);
```

```
    }
  purchaseInfo.reset(product);
  }
}

event LowActivityPurchase(C cart) {
  Set<P>lowActivityProducts = new SetyP>();
  after(P product): LowActivity(product) {
    lowActivityProducts.add(product);
  }
  when(Purchase purchase): RelevantPurchase(purchase) {
  if (lowActivityProducts.contains(purchase.product())) {
    trigger(purchase.cart());
  }
}

aspect LowActivtyDiscount {
  before(C cart): LowActivityPurchase(cart) {
    cart.applydiscount(10);
  }
}
```

Listing 6.17: Bockisch *et al.* event declaration for a low activity discount aspect.

To produce the same results in Chameleon we need to trigger an event when a purchase is made. We assume that there is a purchase method defined for class `Cart`. The goal is to produce an event each time the purchase method is executed. The event `PurchaseEvent` inherits from `MethodExecutionEvent`. So far, no additional condition is defined. The next snippet of code shows the instrumentation of the method `Cart>>purchase`:

```
(EventInstrumentor new) reify: PurchaseEvent
  onClass: Cart selector #purchase
```

Listing 6.18: Reification of a purchase event.

Afterwards, a `DiscountChecker` subscribes to the newly installed event.

```
aChameleonAnnouncer
  subscribe: aDiscountChecker
  to: PurchaseEvent
```

Listing 6.19: Subscription to purchase event.

A discount is applied when the event is triggered and the `DiscountChecker` evaluates that the product involved is eligible for discounts.

So both approaches can deal with this general situation, in which the system is instrumented to check if a discount should be applied to a purchase.

Let us assume that we would like to add a customer benefit system based on purchase points. For every purchase that a customer makes he gets a certain number of points which can be used for future purchases. This benefits point program is optional. Bockisch *et al.* approach would solve this new requirement by adding a new event and aspect. In lines 1–6 the event models when a customer with a points program makes a purchase. In lines 8–12 the aspect keeps the accounting of the customer points depending on the purchase.

```
1  event PointsPurchase(C cart) {
2    when(Purchase purchase): RelevantPurchase(purchase) {
3    if (purchase.customer().hasPointSystem) {
4      trigger(purchase.cart());
5    }
6  }
7
8  aspect CalculatePoints {
9    before(C cart): PointsPurchase(cart) {
10     cart.applypoints();
11   }
12 }
```

Listing 6.20: Aspect solution for a customer points system.

Let us assume that the whole system is now running. There is one customer who did not pay his bills on time. To prevent him from getting more customer points the system should temporarily exclude him from gaining more points. This minor change would force the Bockisch *et al.* approach to change the event's conditions to check for unpaid bills. Every time that there is a constraint change we need to modify by hand the conditions in the events.

Our approach avoids this situation by allowing the user to cherry-pick which objects should produce the events. In this case, we only need to detect when a customer does not pay a bill on time and then remove the instrumentation from his cart. Next time, this particular customer makes a purchase the points system is not triggered.

Chameleon also needs to define a new event modeling when a customer with a points program makes a purchase:

```
1 (EventInstrumentor new) reify: PointsPurchaseEvent
2   onObject: aCart selector: purchase
```

Listing 6.21: Chameleon purchase of a customer with a points program.

With this dynamic instrumentation scoping technique we can dynamically control the scoping of which object should trigger which events, thus preventing the need

to build complicated conditions on the events.

Another issue the Bockisch *et al.* approach solves is controlling whether aspects should be applied to other aspects too. When two or more aspects are woven into a system, and both aspects profile the system, it is not clear if they have to profile each other too. Every AOP approach has to confront this problem. In the Bockisch *et al.* approach the so-called development practices are used to define those entanglements correctly.

```
aspect DevelopmentPractices composes Logger, Proactive, Prevention {
  local declare precedence Logger, Proactive;
  local declare precedence Logger, Prevention;
  local declare overriding Proactive, Prevention;
  local declare ignoring Logger, Proactive;
}
```

Listing 6.22: Development Practice for Logger, Proactive and Prevention

In Chameleon this precedence definition is not required. Which portions of the running system are instrumented to produce events is controlled by applying these instrumentations on top of specific object. Since the instrumentation, observers and events are objects too, in the presence of a new event we can choose which objects should be instrumented. We think in terms of a running system composed of objects.

6.4 Implementation

Chameleon is built on top of the Bifröst reflection framework [Ressia *et al.*, 2010]. Bifröst offers fine-grained unanticipated dynamic structural and behavioral reflection through meta-objects.

6.4.1 Managing AST Meta-Objects

The `EventInstrumentor` is responsible in Chameleon for managing the Bifröst AST meta-objects. An AST meta-object is responsible for adapting the compilation process. These meta-objects are bound to AST nodes which when compiled introduce some extra behavior in the method. When this method is executed the adapted version is run. These meta-objects are transparently managed. The `EventInstrumentor` attaches AST meta-objects to AST nodes to reify different events.

When a node with a meta-object is executed, the meta-object will generate a new event. In our example it is a `MethodExecutionEvent`. The corresponding meta-object creation is seen below:

```
1 ASTMetaObject new
2       delegatingTo: anEvent reificationBlock;
3       arguments: anEvent arguments.
```

Listing 6.23: Bifröst AST meta-object for event reification.

We can see in Listing 6.23 the definition of an AST meta-object. When an AST node is executed and this meta-object is attached to it then the block in line 2 is evaluated with the events arguments in line 3. The event is responsible for providing a reification block which defines how the event is reified and the arguments this block should reify too. Then the event reification method for a single node in the `EventInstrumentor` is defined as follows:

```
1 EventInstrumentor>>reify: anEventClass on: aNode
2    | metaObject |
3    metaObject := ASTMetaObject new
4       delegatingTo: anEventClass reificationBlock;
5       arguments: anEventClass arguments
6    aNode metaObject: metaObject.
```

Listing 6.24: Event reification method for a single node.

In line 6 we associate the AST node to the meta-object.

As an example, let us consider the `MethodExecutionEvent`.

```
1 MethodExecutionEvent class>>reificationBlock
2    ^ [ :selector :class :arguments |
3       MethodExecutionEvent
4          signalMethod: selector
5          class: class
6          arguments: arguments]
```

Listing 6.25: Reification block for reifying the execution of an event.

```
MethodExecutionEvent class>>arguments
   ^ #(selector class arguments)
```

Listing 6.26: Arguments for the reification block for reifying the execution of an event.

We can observe in Listing 6.25 that the reification block only signals the event with parameters defined by `arguments`. The name of the executed method, the class and the arguments are reified together with the event. Every event defines different reification blocks and arguments.

6.4.2 Instrumentation Details

The instrumentation of an event requires several steps. Chameleon provides the behavior to reify an event on all methods of a class.

```
EventInstrumentor>>reify: anEventClass onClass: aClass
   | allNodes reificationNodes |
   self targetClass: aClass.
   self event: anEvent.
   self event adapt: self.
   allNodes := Set new.
   (aClass methodDict keys
      do: [ :key | (aClass>>key) parseTree allChildren
         do: [ :node | allNodes add: node ] ] ).
   reificationNodes := anEventClass
                     reificationNodesIn: allNodes.
   reificationNodes do: [ :node |
                 self reify: anEventClass on: node ].
```

Listing 6.27: Reifying an event for all the methods of a class.

First, the instrumentor needs to find all nodes for all methods of a class. The instrumentor iterates over all methods to obtain all the child nodes including the method node. The decision of which nodes reify the event is delegated to the event itself with the message `reificationNodesIn: nodes`. This message answers a set of nodes that reify the event.

```
ASTNodeExecutionEvent>>reificationNodesIn: aSet
 ^ aSet select: [ :node | self shouldBeReifiedOn: node ]
```

Listing 6.28: Event delegation to decide which AST node reifies a specific event.

In Listing 6.28 we can observe the default implementation of `reificationNodesIn:`. In this method the decision whether a node reifies an event or not is delegated to the event itself through the method `shouldBeReifiedOn: node`.

In the case of the `MethodExectuionEvent` the implementation of `shouldBeReifiedOn:` states that any node that is an AST method node should reify the event that a method is being executed.

```
MethodExectuionEvent>>shouldBeReifiedOn: aNode
   ^ aNode isMethod
```

Listing 6.29: Selection of method nodes for reifying the event that a method was executed.

In the case of an `ASTNodeExecutionEvent` every node inside the chosen scope will be instrumented. For all other events there are different conditions. For `MessageSendEvent` only message send nodes reify this event. `StateReadEvent` is reified by nodes which are variable nodes whose name is also a class instance variable name and are not part of an assignment.

Once the nodes that should reify an specific event are identified the `EventInstrumentor` applies AST node instrumentation on them. This instrumentation adds the necessary behavior to trigger the reified event every time each of these nodes are executed.

Now all method nodes of the bank-account class are reified with method execution events. Every time a bank account receives a message and then the method is executed the `MethodExecutionEvent` will be signaled.

Chameleon also provides behavior for instrumenting specific methods in certain classes.

```
1  EventInstrumentor>>reify: anEventClass onClass: aClass selector: aKey
2     | allNodes reificationNodes |
3     self targetClass: aClass.
4     anEventClasst adapt: self.
5     allNodes := (aClass>>aKey) parseTree allChildren
6                     asSet.
7     reificationNodes := anEventClass
8                     reificationNodesIn: allNodes.
9     reificationNodes do: [ :node |
10                    self reify: anEventClass On: node].
```

Listing 6.30: Reifying an event for a particular method of a class.

Here only all nodes within this particular method are reified.

For example, in the case of the `ObjectCreationEvent` the node on which the event should be reified might not exist. The object creation event depends on the existence of the `new` method. Generally this method is not overridden and is inherited from the superclass. Therefore the method execution event needs the new method to be present in the class. The instrumentor allows the event to adapt the application to add the required node for the reification. This is done through the `adapt:` method which double dispatch through the instrumentor to perform the right adaptation.

```
1  ObjectCreationEvent>>adapt: anEventInstrumentor
2     anEventInstrumentor
3        addMethod: 'new
4        ^ self basicNew.'
5        selector: #new
```

Listing 6.31: Application adaptation for the reification of the method execution event.

This creates a new method (if not already existing) in the target class.

6.4.3 Extending Events

We demonstrate that this approach is more general by implementing Iguana/J's canonical events on top of Chameleon. Therefore we implemented `MessageSendEvent`, `MethodExecutionEvent`, `ObjectCreationEvent`, `ObjectDeletionEvent`, `StateReadEvent` and `StateWriteEvent` on top of `AstNodeExecutionEvent`.

The method dispatch event was not modeled since it can be easily reified using the message send event. Note that the event `MessageReceiveEvent` was also implemented even though it is not part of Iguana/J canonical events. `MessageReceiveEvent` accepts the same nodes as message send but return different arguments for the observers. It signals the node, receiver and the message. This event is particularly useful when modeling message meta-level management like CodA.

6.5 Conclusion

In this chapter we have presented Chameleon, a prototype modeling the meta-level as explicit meta-events observed by development tools. Chameleon realizes a strict separation of concerns between instrumentation and the consumers of events. Moreover, we presented a simplified approach to behavioral reflection through operational decomposition. Our approach proposes a single canonical event on top of which any other object-related event reification can be defined. Event instrumentation can be dynamically applied to specific objects providing better runtime control. By explicitly modeling meta-events the scoping of the development tools can happen at instrumentation time or at event reification time. Our approach provides unpolluted instrumentation, language-unbiased events and dynamic instrumentation scoping.

We demonstrated that object-centric reflection avoids the object paradox in the instrumentation domain. We can develop tools which benefit from seein the system as a set of object-centric dynamic events. This fact provides a simplification of behavioral reflection operational decomposition.

Chapter 7

Profiling Objects

In this chapter we discuss the presence of the object paradox in traditional profiling. We demonstrate that profilers built on top of object-centric reflection avoid this paradox. Moreover, the resulting profiler presents a number of advantages over traditional profilers.

Recent advances in domain-specific languages and models reveal a drastic change in the way software is being built. The software engineering community has seen a rapid emergence of domain-specific tools, ranging from tools to easily build domain-specific languages [Visser, 2004], to transform models [Tisi *et al.*, 2010], to check source code [Renggli *et al.*, 2010a], and to integrate development tools [Renggli *et al.*, 2010c].

While research on domain-specific languages has made consistent progress in language specification [Deursen *et al.*, 2000], implementation [Cuadrado and Molina, 2009], evolution [Freeman and Pryce, 2006] and verification [Kabanov and Raudjärv, 2008], little has been done to support profiling. We consider profiling to be the activity of recording and analyzing program execution. Profiling is essential for analyzing transient run-time data that otherwise would be difficult to harvest and compare. Code profilers commonly employ execution sampling as the way to obtain dynamic run-time information. Unfortunately, information extracted by regularly sampling the call stack cannot be meaningfully used to profile a high-level domain built on top of the standard language infrastructure. Specialized domains need specialized profilers.

Let us consider the example of the Mondrian visualization engine (details follow in Section 7.1.1). Mondrian models visualizations as graphs, *i.e.*, in terms of nodes and edges. One of the important performance issues we recently faced is the refresh frequency: nodes and edges were unnecessarily refreshed too often. Standard code profilers did not help us to localize the source of the problem since they are just able to report the share of time the CPU spends in the method `displayOn:` of the classes `MONode` and `MOEdge`. The problem was finally resolved by developing a custom profiler that could identify which nodes and edges were indeed refreshed too often. This domain-specific profiler was able to exploit knowledge of Mondrian's domain concepts to gather and present the needed information.

We argue that there is a need for a general approach to easily develop specialized profilers for domain-specific languages and tools. A general approach must offer means to (i) *specify* the domain concepts of interest, (ii) *capture* the relevant information from the run-time execution, and (iii) *present* the results to the developer.

In this chapter we detail MetaSpy, an event-based approach for domain-specific profiling. With MetaSpy, a developer specifies the events of interest for a given domain. A profiler captures domain information either by subscribing to existing application events, or by using a reflective layer to transparently inject event emitters into the domain code. The collected events are presented using graph-based visualizations.

The remainder of this chapter is structured as follows: Section 7.1 illustrates the problems of using a general-purpose profiler on code that is built on top of a domain-specific language. Section 7.2 introduces our approach to domain-specific profiling. Section 7.3 demonstrates how our approach solves the requirements of domain-specific profilers with three use cases. Section 7.4 demonstrates how our approach deals with event causality. Section 7.5 presents our infrastructure to implement domain-specific profilers. Section 7.6 presents an analysis on the performance impact of MetaSpy. Section 7.7 summarizes the chapter and discusses future work.

7.1 Shortcomings of Standard Profilers

Current application profilers are useful to gather runtime data (*e.g.*, method invocations, method coverage, call trees, code coverage, memory consumption) from the static code model offered by the programming language (*e.g.*, packages, classes, methods, statements). This is an effective approach when the low-level source code has to be profiled.

However, traditional profilers are far less useful for a domain different than the code model. In modern software there is a significant gap between the model offered by the execution platform and the model of the actually running application. The proliferation of meta-models and domain-specific languages brings new abstractions that map to the underlying execution platform in non-trivial ways. Traditional profiling tools fail to display relevant information in the presence of such abstractions.

MetaSpy[1] and the examples presented in this chapter are implemented in the Pharo Smalltalk[2] programming language, an open-source Smalltalk [Goldberg and Robson, 1983].

1 http://scg.unibe.ch/research/bifrost/metaspy/

2 http://www.pharo-project.org/

7.1.1 Difficulty of profiling a specific domain

This section illustrates three shortcomings of traditional profiling techniques when applied to a specific domain.

CPU time profiling

Mondrian [Meyer *et al.*, 2006] is an open and agile visualization engine. Mondrian describes a visualization using a graph of (possibly nested) nodes and edges. In June 2010 a serious performance issue was raised[3]. Tracking down the cause of the poor performance was not trivial. We first used a standard sample-based profiler.

Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the current set of methods under executions. Such a profiling technique is relatively accurate since it has little impact on the overall execution. This sampling technique is used by almost all mainstream profilers, such as JProfiler, YourKit, xprof [Gupta and Hwu, 1992], and hprof.

MessageTally, the standard sampling-based profiler in Pharo Smalltalk, textually describes the execution in terms of CPU consumption and invocation for each method of Mondrian:

```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
          ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |     | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |     |  2.6% {546ms} FormCanvas>>line:to:width:color:
          ...
   23.4% {4911ms} MOEdge>>displayOn:
         ...
```

We can observe that the virtual machine spent about 54% of its time in the method `displayOn:` defined in the class `MORoot`. A root is the unique non-nested node that contains all the nodes of the edges of the visualization. This general profiling information says that rendering nodes and edges consumes a great share of the CPU time, but it does not help in pinpointing which nodes and edges are responsible for the time spent. Not all graphical elements equally consume resources.

Traditional execution sampling profilers center their result on the frames of the execution stack and completely ignore the identity of the object that received the method call and its arguments. As a consequence, it is hard to track down which objects

3 http://forum.world.st/Mondrian-is-slow-next-step-tc2257050.html#a2261116

cause the slowdown. For the example above, the traditional profiler says that we spent 30.9% in `MONode>>displayOn:` without saying which nodes were actually refreshed too often.

Coverage

PetitParser is a parsing framework combining ideas from scannerless parsing, parser combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically [Renggli *et al.*, 2010b]. A number of grammars have been implemented with PetitParser, including Java, Smalltalk, XML and SQL.

Let us consider a Java grammar in PetitParser which is defined in 210 host language methods. The `if` statement parsing rule is defined as follows:

```
PPJavaSyntax>>ifStatement
   ^ ('if' asParser token , conditionalExpression , statement) ,
     ('else' asParser token , statement) optional
```

These methods build a graph of objects describing the grammar. It would be useful to establish how much of the grammar is actually exercised by a set of test cases to identify untested productions.

Traditional coverage tools focus on the source code artifacts instead of domain-specific data. They assess the coverage of the application source code by listing the methods and source lines covered by an execution.

In our case all methods and all lines of code are covered to build the grammar, but some parts of the resulting graph are not exercised by the tests. This is why we are unable to analyze the parsing and production coverage of this grammar with traditional tools.

Causality

Traditional profilers report events based on the run-time structure of the application. A run-time profiling report is typically structured as a tree in which indentation indicates nested calls. The sequence of methods executed is reported in a linear fashion: A method `m1` that is executed before `m2` will be reported as `m1` above `m2`.

This hardcoded presentation is disconnected from the profiled model. When considering the Mondrian example, the sequence of `displayOn:` methods executed cannot be related to the order in which the nodes are rendered. In PetitParser the order does not represent the sequence in which the parsers are activated.

Understanding the sequence of a large number of events is challenging at best. Unfortunately, textual searching over a log file discards the structure of the model by

solely operating on what the user decided to log. Textual search is a rather limited technique, even though it is commonly employed [Nagappan, 2010].

7.1.2 Requirements for domain-specific profilers

The three examples given above are representative. They illustrate the gap between a particular domain and the source code model. We argue that to efficiently profile an arbitrary domain, the following requirements need to be fulfilled:

- *Specifying the domain.* Being able to effectively designate the objects relevant for the profiling is essential. In Mondrian we are interested in the different nodes and the invocation of the `displayOn:` methods, rather than focusing on the implementing classes. In PetitParser we are interested in how often and if at all production objects are activated by a given input.

- *Capturing domain-related events.* Relevant events generated by the domain have to be monitored and recorded to be analyzed during or after the execution. An event represents a particular change or action triggered by the domain being profiled. Whereas the class `MOGraphElement` and its subclasses total more than 263 methods, only fewer than 10 methods are related to displaying and computing shape dimensions.

- *Effectively and concisely presenting the necessary information.* The information collected by traditional profilers is textual and targets method invocation. A method that invokes another will be located below it and indented. Moreover, each method frame represented has a class name and a method name, which completely ignores the identity of the object and arguments that are part of the call. Collected information has to be presented in such a way as to bring the important metrics and domain object composition into the foreground.

- *Relation between events.* An important and recurrent task in profiling is to understand the meaning of a sequence of emitted events. This is necessary when a developer wants to understand the causes of a suboptimal execution. Captured events have to be causally related to each other to trace high level operations. Since such relation between events cannot be enforced by the domain, it has to be reconstructed upon reception. Captured events have to be presented in a sequence that reflects the meaning of the model operations.

- *Browsing events.* The number of events generated by a typical application execution may easily skyrocket. Diving into those events is often the only way to understand the reason for suboptimal execution. Navigating through and giving a meaning to such a large number of events requires adequate tools that are aware of the model used to generate the events.

Common code profilers employ execution sampling as the way to cheaply obtain dynamic information. Unfortunately, information extracted when regularly sampling the method call stack cannot be used to profile a domain other than the source code model.

7.2 MetaSpy in a Nutshell

In this section we will present MetaSpy, a framework that supports building domain-specific profilers. The key idea behind MetaSpy is to provide domain-specific events that can later be used by different profilers with different objectives.

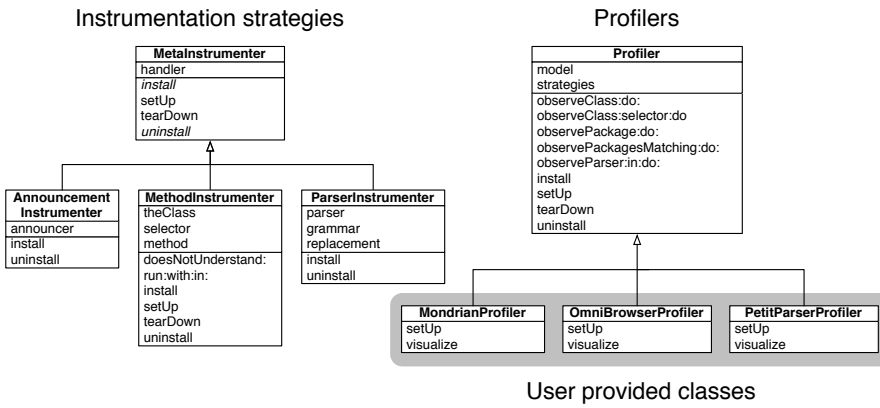

Figure 7.1: The architecture of the MetaSpy profiler framework.

Figure 7.1 shows a class diagram of MetaSpy. There are two main abstractions: the instrumentation strategies and the domain-specific profilers.

An instrumentation strategy is responsible for adapting a domain-specific model and triggering specific actions in the profiler when certain events occur. A profiler models a domain-specific profiling requirement by composing multiple instrumentation strategies.

Some instrumentation strategies work by registering to existing events of the application domain. Other instrumentation strategies intercept the system by metaprogramming, *i.e.*, conventional instrumentation. Installing an instrumentation strategy activates it and its associated events, while uninstalling deactivates them.

Some of the instrumentation strategies provided by MetaSpy are:

- *Announcement Instrumenter* dispatches events satisfying a particular condition from the announcer (subject) to the external profiler (observer).

- *Method Instrumenter* triggers an event whenever a specific method is invoked on any instance of a specified class.
- *Object Instrumenter* triggers an event whenever a specific method is invoked on a particular object. This is called object-specific profiling.
- *Parser Instrumenter* triggers an event whenever a specific grammar production is activated. This is a very specific instrumentation strategy only working with PetitParser productions.

Other dedicated instrumentation strategies can be implemented by adhering to the same interface.

Profilers are responsible for modeling the domain-specific behavior to profile the main abstractions into each domain. The abstract `Profiler` class models the behavior of a general profiler. Subclasses are instantiated with a domain-specific model and implement the set-up and tear-down of one or more instrumentation strategies into the model. Furthermore, they define how and what data is collected when the instrumented model is exercised. To actually instrument the model and start collecting events the method `install` is used. Similarly, to remove all instrumentation from the model, `uninstall` is used. Both methods dispatch the requests to the respective instrumentation strategies using the current model.

Each profiler is responsible for presenting the collected data in the method `visualize`. Depending on the nature of the data, this method typically contains a Mondrian [Meyer *et al.*, 2006] or Glamour [Bunge, 2009] script, or a combination of both. Mondrian is a visualization engine to depict graphs of objects in configurable ways. Glamour is a browser framework to script user interfaces for exploratory data discovery.

Next, we will show real-world examples of domain-specific profilers.

7.3 Validation

In this section we will analyze three case studies from three different domains. We will show how MetaSpy is useful for expressing the different profiling requirements in terms of events. We will also demonstrate how MetaSpy fulfills the domain-specific profiling requirements, namely specifying, capturing, and presenting domain-specific information.

For each case study we show the complete code for specifying and capturing events. We do not show the code for visualizing the results, which typically consists of 20–50 lines of Mondrian or Glamour script code. We use the Mondrian visualization tool

to visually and interactively report profiles. In Section 7.3.1 we also consider Mondrian as the profiling subject. We therefore visualize using Mondrian the profile of Mondrian itself.

7.3.1 Case Study: Displaying invocations

A Mondrian visualization may comprise a great number of graphical elements. A refresh of the visualization is triggered by the operating system, resulting from user actions such as a mouse movement or a keystroke. Refreshing the Mondrian canvas iterates over all the nodes and edges and triggers a new rendering. Elements that are outside the window or for which their nesting node has an active bitmap in the cache should not be rendered.

A graphical element is rendered when the method `display:on:` is invoked. Monitoring when these invocations occur is key to having a global view of what should be refreshed.

Capturing the events

The MetaSpy framework is instantiated to create the `MondrianProfiler` profiler.

```
Profiler subclass: #MondrianProfiler
  instanceVariableNames: 'actualCounter previousCounter'
```

`MondrianProfiler` defines two instance variables to monitor the evolution of the number of emitted events: `actualCounter` keeps track of the current number of triggered events per event type, and `previousCounter` stores the number of event types that were recorded before the previous visualization step.

```
MondrianProfiler>>initialize
   super initialize.
   actualCounter := IdentityDictionary new.
   previousCounter := IdentityDictionary new
```

The installation and instrumentation of Mondrian by MetaSpy is realized by the `setUp` method:

```
MondrianProfiler>>setUp
   self model root allNodes do: [ :node |
      self
         observeObject: node
         selector: #displayOn:
         do: [ :receiver :selector :arguments |
            actualCounter
```

```
            at: receiver
            put: ((actualCounter at: receiver ifAbsent: [ 0 ]) + 1) ] ]
```

All the nodes obtained from the root of the model object are "observed" by the framework. At each invocation of the `displayOn:` method, the block given as parameter to `do:` is executed with the object receiver on which `displayOn:` is invoked, the selector name and the argument. This block updates the number of displays for each node of the visualization.

Specifying the domain

The instrumentation described in the `setUp` method is only applied to the model specified in the profiler. This model is an object which models the domain to be profiled, in this case a Mondrian visualization. The instrumentation is only applied to all nodes in this visualization. Only when these nodes receive the message `displayOn:`, the actual counter is incremented. This object-specific behavior is possible due to the use of Bifröst [Ressia *et al.*, 2010] meta-objects.

Presenting the results

The profiling of Mondrian is visualized using Mondrian itself. The `visualizeOn:` method generates the visualization given in Figure 7.2.

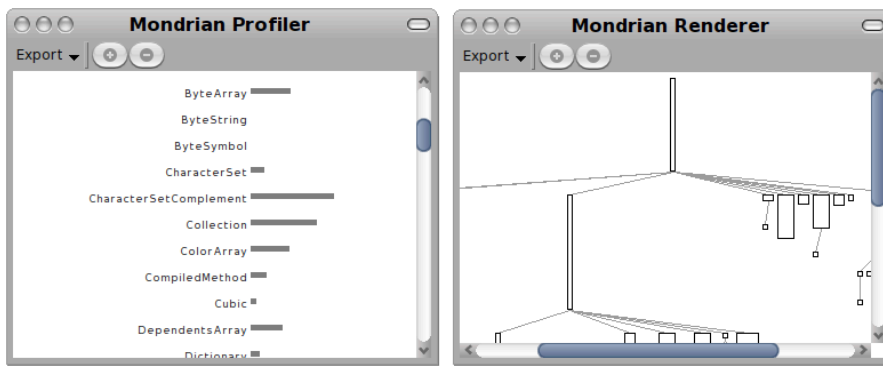

Figure 7.2: Profiling (left) the System Complexity visualization (right).

One important point of `visualizeOn:` is to regularly update the visualization to be able to see the evolution of the domain events over time.

Figure 7.2 gives a screenshot of a visualization and the profiler. The right-hand side is an example of the *System Complexity* visualization [Lanza and Ducasse, 2003] of

the collection class hierarchy. System complexity is a typical usage of Mondrian, which exhibits the problem mentioned in Section 7.1.1.

The left-hand side shows the profiler applied to the visualization on the right-hand side. The profiler lists all the classes visualized in the system complexity. The profiler associates to each class a horizontal bar indicating the number of times the corresponding node in the system complexity has been displayed. This progress bar widens upon node refresh. The system complexity visualization remains interactive, even when being profiled. Selecting, dragging and dropping nodes refreshes the visualization, thus increasing the displayed progress of the corresponding nodes. This profile helps in identifying unnecessary rendering. Thanks to this profiler, we identified a situation in which nodes were refreshing without receiving user actions which caused the sluggish rendering. More precisely, edges were constantly refreshed, even when they were not visible. The profiler is uninstalled when the profiled Mondrian visualization is closed.

7.3.2 Case Study: Events in OmniBrowser

OmniBrowser [Bergel *et al.*, 2008] is a framework for defining and composing new browsers, *i.e.*, graphical list-oriented tools to navigate and edit elements from an arbitrary domain. In the OmniBrowser framework, a browser is described by a domain model specifying the domain elements that can be navigated and edited, and a metagraph specifying the navigation between these domain elements. Nodes in the metagraph describe states the browser is in, while edges express navigation possibilities between those states. The OmniBrowser framework then dynamically composes widgets such as list menus and text panes to build an interactive browser that follows the navigation described in the metagraph.

OmniBrowser uses announcements for modeling the interaction events of the user with the IDE. A very common problem is to have certain announcements be triggered too many times for certain scenarios. This behavior impacts negatively the performance of the IDE. Moreover, in some cases odd display problems are produced which are very hard to track down.

Capturing the events

To profile this domain-specific case we implemented the class `OmniBrowserProfiler`:

```
Profiler subclass: #OmniBrowserProfiler
   instanceVariableNames: 'actualCounter'
```

The instrumentation in the `setUp` method counts how many times each announcement was triggered.

```
OmniBrowserProfiler>>setUp
   self
      observeAnnouncer: self model announcer
      do: [ :ann |
         actualCounter
            at: ann class
            put: (actualCounter at: ann class ifAbsent: [ 0 ]) + 1 ]
```

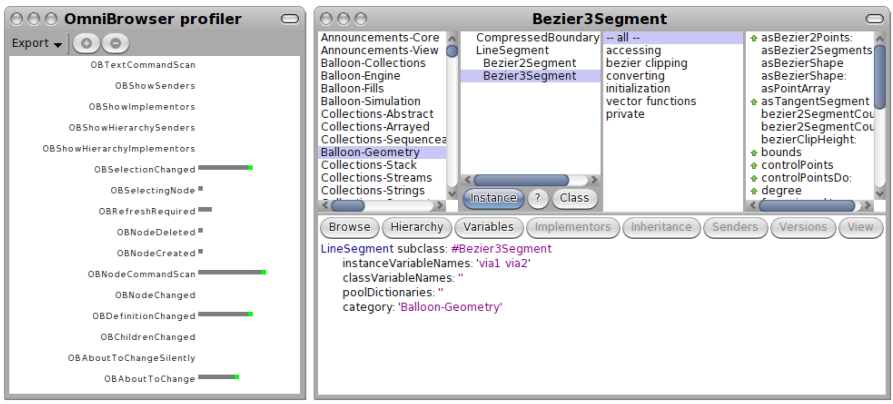

Figure 7.3: Profiling (left) an OmniBrowser instance (right).

Specifying the domain

We specify the entities we are interested in profiling by defining the model in the profiler. The model is an instance of the class `OBSystemBrowser`, the entry point of OmniBrowser. All OmniBrowser instances have an internal collaborator named *announcer* which is responsible for the signaling of announcements. This is the object used by the profiler to catch the announcement events.

Presenting the results

A Mondrian visualization was implemented to list the type and the number of announcements triggered (cf. Figure 7.3).

7.3.3 Case Study: Parsing framework with PetitParser

Rigorous test suites try to ensure that each part of the grammar is covered by tests and is well-specified according to the respective language standards. Validating

that each production of the grammar is covered by the tests is a difficult activity. As mentioned previously, traditional tools of the host language work at the method and statement level and thus cannot produce meaningful results in the context of PetitParser where the grammar is modeled as a graph of objects.

Capturing the events

With MetaSpy we can implement the grammar coverage with a few lines of code. The instrumentation happens at the level of the primitive parser objects. The method `observeParser:in:` wraps the parser object with a handler block that is called for each activation of the parser.

```
1 PetitParserProfiler>>setUp
2    self model allParsers do: [ :parser |
3       self observeParser: parser in: self grammar do: [
4          counter
5             at: parser
6             put: (counter at: parser ifAbsent: [ 0 ]) + 1 ] ]
```

Line 2 iterates over all primitive parser objects in the grammar. Line 3 attaches the event handler on Lines 4–6 to each parser in the model. The handler then counts the activations of each parser object when we run the test suite of the grammar.

Specifying the domain

The domain in this case is an instance of the grammar that we want to analyze. Such a grammar may be defined using hundreds of interconnected parser objects.

Presenting the results

This provides us with the necessary information to display the grammar coverage in a visualization such as that shown in Figure 7.4.

7.4 Identifying Event Causality

Mondrian visualizes graphs of nodes and edges. Apart from the edges displayed in the visualization, nodes can support other relationships: nodes might be nested within each other, *i.e.*, when a parent is moved, its children have to be redrawn; nodes might have interaction dependencies, when one node is selected another one is updated; nodes might have caching dependencies, when one node changes dependent nodes need to invalidate their state; and so on.

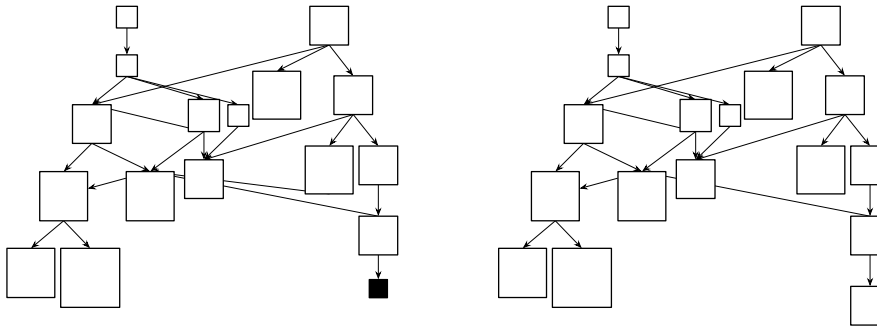

Figure 7.4: Visualization of the production coverage of an XML grammar with uncovered productions highlighted in black (left); and the same XML grammar with updated test coverage and complete production coverage (right). The size of the nodes is proportional to the number of activations when running the test suite on the grammar.

The use of log files to identify such dependencies may indeed be successful [Yaghmour and Dagenais, 2000]. However, producing an adequate log file that covers all the different situations requires a significant amount of work and good system knowledge.

We favor prototyping of lightweight tools to address the possible problems on the spot.

7.4.1 Expressing causality

According to the experience we gain by profiling multiple model executions, events generated by the model cannot be used to meaningfully structure an execution profile. This is not really a surprise since events are generated from a model to fulfill a need of the model itself, and not really for profiling purposes. No assumption can therefore be made on the information carried by those events.

A practical solution is to annotate events upon reception with information about the sequentiality and the timing. MetaSpy offers a generic event class, called `SpyEvent`. A spy event knows its creation time and the previously emitted event.

The class `SpyEvent` may be subclassed to capture domain relations. For example, `MondrianEvent` knows about siblings of the node that emitted the event. This is an important relation for tracing how the cache is activated.

7.4.2 Navigation between events

To analyze the event activation sequence in Mondrian we have the following spy:

```
Profiler subclass: #MSMondrianCacheActivationSequenceProfiler
   instanceVariableNames: 'lastEvent mapping announcer'
```

This profiler has three variables. The last event that has been emitted is kept in the variable `lastEvent`. Since all the events are kept in a linked list, it is sufficient to keep a reference of the last event to access previous events. The association between a Mondrian node and the events the node has emitted is kept in the variable `mapping`. The browser is updated via an announcer.

The profiler is installed with `setUp`:

```
MondrianCacheActivationSequenceSpy>>setUp
   super setUp.
   nodes do: [ :node |
      self
         observeObject: node
         selector: #displayOn:
         do: [ :receiver :selector :arguments |
            lastEvent := (MondrianEvent for: receiver next: lastEvent).
            (mapping at: receiver ifAbsentPut: [ OrderedCollection new ])
               add: lastEvent ] ]
```

`MondrianCacheActivationSequenceSpy` is responsible for adapting Mondrian nodes to find out the order in which the method `displayOn:` was executed. Each execution of the method `displayOn:` should create an instance of `MondrianEvent`. Each Mondrian node is instrumented so that every time that the message `displayOn:` is invoked a `MondrianEvent` is created and saved within the mappings indexed by node. Each `MondrianEvent` knows the node that generated it and the previous event. The `setUp` is invoked to install the instrumentation.

Mondrian events are first captured during the profile. The browsing tool described below is useful to navigate between them.

Glamour [Bunge, 2009] is an engine for scripting browsers. We use it to build navigation tools for the captured events. The Glamour-based tool is set up in the `visualize` method:

```
MondrianCacheActivationSequenceSpy>>visualize
   | browser |
   browser := Tabulator new.
   browser title: 'Mondrian event crawler'.
   browser
      column: #events;
      column: #model.
   browser transmit to: #events;
      andShow: [ :constructor | self eventsIn: constructor ].
```

```
browser transmit from: #events; to: #model;
    andShow: [ :constructor | self modelIn: constructor ].
browser updateOn: Announcement from: [ :v | announcer ].
browser openOn: lastEvent.
```

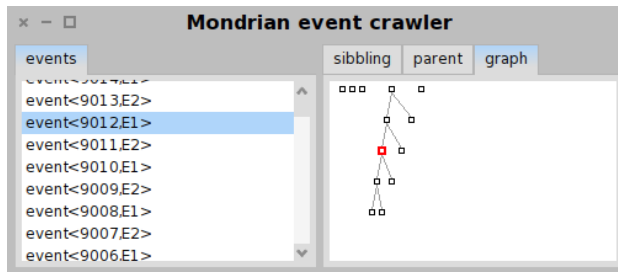

Figure 7.5: Glamour-based event navigation tool.

Figure 7.5 shows the result of a Mondrian profiling using the Glamour script. The left-hand side gives the sequential list of the events we captured using emitted by Mondrian. The right-hand side gives the information associated to the event selection.

The method `modelIn:` is invoked when one selects an event. The method fills a glamour element with three tabs, two lists and a Mondrian visualization:

```
MondrianCacheActivationSequenceSpy>>modelIn: constructor
modelIn: constructor
   constructor list
      title: 'sibling';
      display: [ :event | self siblingOf: event ].
   constructor list
      title: 'parent';
      display: [ :event | self parentOf: event ].
   constructor mondrian
      title: 'graph';
      painting: [ :view :event |
         view nodes: (self siblingOf: event).
         view edgesFrom: #owner.
         view treeLayout ].
```

The methods `parentOf:` and `siblingOf:` are used to retrieve the data from the Mondrian model and are not presented here.

The list of events are accessed using the helper method:

```
MondrianCacheActivationSequenceSpy>>eventsIn: constructor
   constructor list
```

```
    title: 'events';
    display: [ :event | event allPreviousEvents ];
    updateOn: Announcement from: [ :v | announcer ]
```

Events are linked to each other forming a list. The method `allPreviousEvents` returns the list of all previous events.

Using an adequate model, a browsing tool is easily implementable using Glamour. Presentations are constructed and combined to reflect the navigation flow of the extracted events.

The variable `mapping` plays an important role since it associates the events with the node who emitted them. A hash map effectively implements this relation.

7.5 Implementing Instrumentation Strategies

MetaSpy has two ways of implementing instrumentation strategies: listening to pre-existing event-based systems, or using the meta-level programming techniques of the host language to define a meta-event the strategy is interested in.

Let us consider the class `AnnouncementInstrumenter`, whose responsibility is to observe the generation of specific announcements.

```
AnnouncementInstrumenter>>install
   self announcer
      on: Announcement
      send: #value:
      to: self handler
```

The `install` method installs an instrumentation strategy object on the domain specified in the `install` method. In this snippet of code we can see that the strategy is hooked into the announcement system by evaluating the strategy's handler when an announcement is triggered.

However, not all profiling activities can rely on a pre-existing mechanism for registering to events. In some cases, a profiler may be hooked into the base code using an existing event mechanism, for example the OmniBrowser profiler. In other cases, extending the base code with an appropriate event mechanism is simply too expensive. Because of this, we need to rely on the meta-programming facilities of the host language. These facilities are not always uniform and require ad hoc code to hook in behavior. To avoid this drawback we decided to use a framework that provides uniform meta-programming abstractions.

7.5.1 Bifröst

MetaSpy instrumentation mechanism is built on top of Bifröst meta-objects. Let us consider the Message Received Instrumenter, whose responsibility is to instrument when a specific object receives a specific message.

```
MessageReceivedInstrumenter>>install
   self observerMetaObject bind: self object
```

```
MessageReceivedInstrumenter>>setUp
   profilingMetaObject := BehaviorMetaObject new
      when: self selector
      isReceivedDo: self handler
```

The method `install` binds a meta-object to the object to be observed. The method `setUp` initializes the profiling meta-object with a behavioral meta-object. This meta-object evaluates the handler when a specific message is received by the profiled object. This mechanism is termed *object-specific instrumentation*.

In our Smalltalk implementation of Bifröst, the profiled application, the profiler, and the visualization engine are all written in the same language, Pharo, and run on the same virtual machine. Nothing in our approach prevents these components from being decoupled and having them written in a different language or running remotely. This approach is often taken with profilers and debuggers running on the Java virtual machine (*e.g.*, Java debugging interface[4]).

7.5.2 Feasibility of Domain-specific Profiling

Let us analyze the feasibility of implementing this approach in other contexts. Object-specific instrumentation is not trivial to achieve in class-based languages like Smalltalk and Java. Classes are deeply rooted in the language interpreter or virtual machine and performance is tweaked to rely heavily on these constructs. Moreover, most languages provide a good level of structural reflection to deal with structural elements like classes, method, statements, *etc.* Most languages, however, do not provide a standard mechanism to reflect on the dynamic abstractions of the language. There are typically no abstractions to intercept meta-events such as a message send, a message receive, a state read, *etc.* There has recently been extensive work on object-specific runtime adaptations and operation decomposition of the runtime system.

Aspect-Oriented Programming (AOP) [Kiczales, 1996; Kiczales *et al.*, 1997b; Kiczales *et al.*, 1997a] is a technique which aims at increasing modularity by supporting the separation of cross-cutting concerns. Dynamic object-specific aspects have been introduced with an operational decomposition view of the system.

4 `http://download.oracle.com/javase/1.5.0/docs/guide/jpda/jvmdi-spec.html`

Douence, Motelet and Südholt [Douence *et al.*, 2001] introduced a general operational model for crosscutting based on execution monitors called Event-based Aspect-Oriented Programming (EAOP). Douence and Südholt [Douence and Südholt, 2002] later introduced constructor calls and constructor returns as events. The Execution Monitor in their implementation observes events emitted during execution. The execution of the base program is suspended when an event is emitted. The monitor matches this event against different event patterns. When a pattern is satisfied the associated actions are executed.

The JAsCo language [Suvée *et al.*, 2003] provides a way of separating *when* an aspect should be applied and *what* should be done. Hooks are defined with abstract pointcuts. Traps are introduced in the potential places where an event should be triggered. Connectors link these events to the hooks that dictate what should be done. Connectors can be loaded dynamically making this approach highly dynamic.

Stateful aspects or tracematches make it possible to restrain the application of an aspect to the occurrences of certain execution event patterns. AspectJ extension with tracemath [Allan *et al.*, 2005] events patterns are matched in all threads of the system.

Domain-specific profiling can be achieved using other techniques. However, the biggest difference is the unanticipation present in MetaSpy which is hard to achieve in other approaches like AOP, EAOP and dynamic aspects. Nonetheless, it is possible to preplan, before running, which portions of the application might be adapted for profiling thus achieving a very similar approach to MetaSpy.

7.6 Micro-benchmark

Profiling always impacts the performance of the application being analyzed. We have performed a micro-benchmark to assess the maximal performance impact of MetaSpy. We assume that the behavior required to fulfill the profiling requirements is constant to any instrumentation strategy.

We analyze the impact of MetaSpy on both profiling uses cases. All benchmarks were performed on an Apple MacBook Pro, 2.8 GHz Intel Core i7 in Pharo 1.1.1 with the jitted Cog VM.

Registering instrumentation strategies to a pre-existing event-based system depends heavily on the the system used and how it is used.

Using meta-level programming techniques on a runtime system can have a significant performance impact. Consider a benchmark in which a test method is being invoked one million times from within a loop. We measure the execution time of the benchmark with Bifröst reifying the 10^6 method activations of the test method. This shows that in the reflective case the code runs about 35 times slower than in the reified one. However, for a real-world application with only few reifications the

performance impact is significantly lower. Bifröst's meta-objects provide a way of adapting selected objects thus allowing reflection to be applied within a fine-grained scope only. This provides a natural way of controlling the performance impact of reflective changes.

Let us consider the Mondrian use case presented in Section 7.1.1. The main source of performance degradation is from the execution of the method `displayOn:` and thus whenever a node gets redisplayed. We developed a benchmark where the user interaction with the Mondrian easel is simulated to avoid human delay pollution in the exercise. In this benchmark we redraw one thousand times the nodes in the Mondrian visualization. This implies that the method `displayOn:` is called extensively. The results showed that the profiler-oriented instrumentation produces on average a 20% performance impact. The user of this Mondrian visualization can hardly detect the delay in the drawing process. Note that our implementation has not been aggressively optimized. It has been shown [Arnold and Ryder, 2001] that combining instrumentation and sampling profiling leaded to accurate profiles (93–98% overlap with a perfect profile) with low overhead (3–6%). The profilers we presented in this chapter are likely to benefit from such instrumentation sampling.

7.7 Conclusions

We demonstrated the need for domain-specific profilers. We argued that traditional profilers present the object paradox since they are concerned with source code only and are inadequate for profiling domain-specific concerns. We demonstrated this drawback with two use cases. Moreover, we demonstrated how the reflective requirements are present in Bifröst unified reflection approach. We formulated the requirements domain-specific profilers must fulfill: specifying the domain, capturing domain related events and presenting the necessary information. We presented MetaSpy, a framework for defining domain-specific profilers. We also presented three real-world case studies showing how MetaSpy fulfills the domain-specific profiler requirements.

Chapter 8

Object-Centric Debugging

In this chapter we demonstrate the presence of the object paradox in traditional debuggers. We demonstrate that debuggers built on top of object-centric reflection avoid the paradox. Moreover, the resulting debugger closes the gap between the developer questions and the debugging operations thus speeding the debugging and bug fixing process.

During the process of developing and maintaining a complex software system, developers pose detailed questions about the runtime behavior of the system. Source code views offer strictly limited insights, so developers often turn to tools like debuggers to inspect and interact with the running system. Unfortunately, traditional debuggers focus on the runtime stack as the key abstraction to support debugging operations, though the questions developers pose often have more to do with objects and their interactions.

Sillito *et al.* [Sillito *et al.*, 2006] identified 44 kinds of questions that programmers ask themselves when they perform a change task on a code base. A typical such question which is particularly relevant here is: *Where is this variable or data structure being accessed?* Developers take two approaches to answer this question. The first approach is to follow the control flow and use the *step over* and *step into* stack-based operations. Manual step-wise execution works well when the code space to explore is relatively small, but may be impractical otherwise. The second approach is to place breakpoints in all potential places where the variable might be accessed. Again, this can work well for a small code space, but can quickly become impractical if a variable is potentially accessed from many methods. Some debuggers allow the developer to insert breakpoints on accesses to instance variables. However, when such a breakpoint is applied to a particular class all instances of the class are affected. If the developer needs to follow a specific object's instance variable access, then he needs to proceed through breakpoint executions until the right object is found. Even with a small number of instances this process is error prone and not straightforward.

These approaches are inherently *static* since they start from the static source code. Neither approach directly answers the question *"Where is this variable or data structure being accessed?"* for a specific object. There is consequently a gap between the kinds

of questions developers ask about the running software system and the support offered by traditional debuggers to answer these questions.

Object-centric debugging attempts to close the gap between developers' questions and the debugging tool by shifting the focus in the debugger from the execution stack to individual objects. The essence of object-centric debugging is to let the user perform operations directly on the objects involved in a computation, instead of performing operations on the execution stack. A fundamental difference between conventional and object-centric debugging is that the latter is specified on an *already running program*. Instead of setting breakpoints that refer to source code, one sets breakpoints with reference to a particular object.

This chapter is structured as follows: Section 8.1 explains and motivated the need for object-centric debugging. Section 8.2 presents the object-centric approach with its operations. In Section 8.3 we demonstrate how our approach solves the challenges of object-centric debugging with various case studies. Section 8.4 presents our infrastructure to implement object-centric debuggers. Section 8.5 analyzes how object-centric debugging can be implemented in other languages. In Section 8.6 we discuss the state of the art of debugging. Section 8.7 summarizes this chapter and discusses future work.

8.1 Motivation

During software development and evolution, programmers typically need answers to various questions about how the software behaves at runtime. Although various dynamic analysis tools exist, the programmers' first mainstream tool choice to explore the state of a running program is the debugger. The classical debugger requires the programmer to set breakpoints in the source code before debugging is enabled, and then offers the programmer operations to explore the execution stack. Unfortunately the debugger is not designed to answered many of the questions that programmers typically pose, making it difficult, if not impossible for the programmer to set meaningful breakpoints.

In this section we explore these questions, and establish three challenges that a debugger should meet to better support software evolution tasks, namely: (i) intercepting access to object-specific runtime state; (ii) monitoring object-specific interactions; and (iii) supporting live interaction. These challenges lead us to propose *object-centric debugging* to meet these challenges.

8.1.1 Questions Programmers Ask

Sillito *et al.* [Sillito *et al.*, 2006] identified 44 kinds of questions that programmers ask when they perform a change task on a code base. Several of these questions involve understanding the program execution:

- *When during the execution is this method called? (Q.13)*
- *Where are instances of this class created? (Q.14)*
- *Where is this variable or data structure being accessed? (Q.15)*
- *What are the values of the argument at runtime? (Q.19)*
- *What data is being modified in this code? (Q.20)*
- *How are these types or objects related? (Q.22)*
- *How can data be passed to (or accessed at) this point in the code? (Q.28)*
- *What parts of this data structure are accessed in this code? (Q.33)*

Sillito *et al.* note: "In several sessions, the debugger was used to help answer questions of relevancy. Participants set breakpoints in candidate locations (without necessarily first looking closely at the code)." In the context of a running object-oriented system these questions express that programmers need to deal with *specific objects* at runtime.

Consider, for example, questions 13 and 14. Simply by placing a breakpoint in the method concerned (Q.13), or in the constructor(s) of the class being instantiated (Q.14), and running either the program or its test suite one can quickly obtain answers to these questions and then explore the execution stack to obtain detailed information about the calling context.

This procedure works fine when trying to understand the general behavior of objects. However, when introducing polymorphism and delegation the behavior of objects of the same class changes depending on their composition. These cases require an object-specific analysis and simple breakpoint are not the best option. Conditional breakpoints are heavily used in real world application development when programmers need to interrupt the execution of the application when a particular expression is evaluated to true. First the programmer needs to find the specific object he is interested in. Then the programmer has to specify a suitable condition to identify the specific object already found, rather than directly interacting with it. This approach may be feasible if there exist only few objects to analyze. If, however, there are many instances of many classes, setting conditional breakpoints may be tedious and error prone. We present examples of these shortcomings in Section 8.1.3, Section 8.1.4 and Section 8.1.5.

The situation is much the same with many of the other questions.

8.1.2 Getting to the Objects

Both traditional stack-centric and object-centric debuggers share a common operational process. Developers use debuggers to understand the runtime behavior of a system. In the runtime the developer deals with objects instead of with their static representation in the source code. Stack-centric and object-centric debugging diverge when the developer finds a particular object that is not behaving as expected. In a traditional stack-centric debugger the developer leaves the debugger and turns to the static representation of the system to place regular or conditional breakpoints to steer the execution around a particular object. In an object-centric debugger the developer does not need to leave the debugger but applies object-centric operations directly on the object of interest.

In both cases we need to get to the objects that are relevant for the debugging case. However, once the relevant objects have been detected, the steering method is completely different. Object-centric debugging allows the developer to continue interacting with the runtime, applying operations directly on the objects, instead of working with the static representation of the system.

8.1.3 Intercepting Object-specific State Access

Questions 15, 19, 20, 28 and 33 all have to do with tracking state at runtime. Consider in particular question 15: *Where is this variable or data structure being accessed?* Let us assume that we want to know where an instance variable of an object is being modified. This is known as keeping track of side-effects [Maruyama and Terada, 2003]. One approach is to use step-wise operations until we reach the modification. However, this can be time-consuming and unreliable. Another approach is to place breakpoints in all assignments related to the instance variable in question. Finding all these assignments might be troublesome depending on the size of the use case, as witnessed by our own experience.

During the development of a reflective tool we faced the situation that an unexpected side effect occurred. The bytecode interpreter of the host language[1] is modeled by the class `InstructionStream`. This class defines an instance variable called `pc` (*i.e.*, program counter) which models where the execution is in the instruction stream. The class `MethodContext` is a subclass of `ContextPart`, itself a subclass of `InstructionStream`. During our development, we encountered an unexpected increase of the variable `pc` in an instance of `MethodContext`. Tracking down the source of this side effect is highly challenging: 31 of the 38 methods defined on `InstructionStream` access the variable, comprising 12 assignments; the instance variable is written 9 times in `InstructionStream`'s subclasses. In addition, the variable `pc` has an accessor that is referenced by 5 intensively-used classes. Without a deep understanding of

1 `http://www.pharo-project.org/`

the interpreter, it is difficult to track down the source of the error with simple debugging strategy. Some debuggers provide instance variable related breakpoints. However, these breakpoints are not object-specific thus requiring the introduction of conditional breakpoints to interrupt execution only in the right context.

Questions 19, 20, 28 and 33 can also be difficult to answer through classical debugging. The typical approach in each case is to *statically* identify possible call sites that may access or modify the data in question, insert breakpoints, and then invoke the debugger. For complex programs (which are the only programs that are really of interest), finding and setting suitable breakpoints may be an overwhelming task, and running the debugger may yield false positives.

8.1.4 Monitoring Object-specific Interactions

Let us reconsider question 13: *When during the execution is this method called?* If the programmer is only interested in knowing when the method is called *for a specific object* (or caller), then a *conditional breakpoint* may be set, *i.e.*, which will only cause the debugger to start if the associated condition is met. This, however, assumes that the object can be statically identified, since the breakpoint is set in the source code view, not at runtime. Furthermore, if the source code of the object in question is not accessible, the programmer will be forced to set breakpoints at the call sites.

Question 22 poses further difficulties for the debugging approach: *How are these types or objects related?* In statically typed languages this question can be partially answered by finding all the references to a particular type in another type. Due to polymorphism, however, this may still yield many false positives. (An instance variable of type `Object` could be potentially bound to instances of any type we are interested in.) Only by examining the runtime behavior can we learn precisely which types are instantiated and bound to which variables. The debugging approach would, however, require heavy use of conditional breakpoints (to filter out types that are not of interest), and might again entail the setting of breakpoints in a large number of call sites.

8.1.5 Supporting Live Interaction

Back-in-time debugging [Lewis, 2003; Pothier *et al.*, 2007] can potentially be used to answer many of these questions, since it works by maintaining a complete execution history of a program run. There are two critical drawbacks, however, which limit the practical application of back-in-time debugging. First, the approach is inherently *post mortem*. One cannot debug a running system, but only the history of a completed run. Interaction is therefore strictly limited, and extensive exploration may require many runs to be performed. Second, the approach entails considerable overhead both in terms of runtime performance and in terms of memory requirements to build and explore the history.

Although conventional debugging is more interactive, it also requires much advance preparation in terms of exploring the static source code to set breakpoints of potential interest. As a consequence, also conventional debuggers fall short in supporting live, interactive debugging.

8.1.6 Towards Object-Centric Debugging

If we reexamine the set of questions identified by Sillito *et al.* that relate to running software, we can see that they essentially cover all possible combinations of: *"From where and what is this object's state accessed?"* and *"How does this object interact with other objects?"* In other words, the focus of programmers' questions appears to be not the execution stack but rather the *objects* in the running system.

We therefore hypothesize that an *object-centric debugger* — *i.e.*, a debugger that allows one to set breakpoints on access to individual objects, to its methods and to its state — might better support programmers in answering typical development questions. In particular an object-centric debugger would (i) intercept object-specific state access without needing one to set breakpoints in call sites or state. (ii) monitor interactions with individual objects without requiring conditional breakpoints; and (iii) support lightweight, live interaction with a running system without requiring breakpoints in source code.

8.2 Object-Centric Debugging

8.2.1 Object-Centric Debugging in a Nutshell

Conventional debugging allows one to interrupt and interact with a running program by specifying breakpoints in the execution flow of the program. *Object-centric debugging*, by contrast, interrupts execution when a given object is accessed or modified. Whereas conventional debugging requires breakpoints to be set at locations corresponding to points in the source code, object-centric debugging intercepts interactions that do not necessarily correspond to specific points in the source code.

As we saw in the previous section, of the questions that programmers pose about software, the most problematic ones are those dealing with how and where the state of an object is accessed, and how an object interacts with other objects. Object-centric debugging therefore introduces mechanisms to intercept execution on precisely those interactions.

A fundamental difference between conventional and object-centric debugging is that the latter is specified on an *already running program*. Instead of setting breakpoints that refer to source code, one sets breakpoints with reference to a particular object. This means that object-centric debugging operations can only be applied to a

running program that has already been interrupted, possibly with the help of a conventional breakpoint. Clearly this implies that object-centric debugging is intended to augment conventional debugging, not to replace it.

Let us see which object-centric debugging operations are supported.

8.2.2 State-related operations

There are two object-centric debugging operations that intercept accesses to object state.

Halt on write. When an instance variable of an object is changed the execution should be halted. We can scope this operation to any instance variable of the object or to a particular one.

Halt on read. The execution is halted when an object's instance variable is used. We can scope this operation to any instance variable or to a specific one.

8.2.3 Interaction operations

There are six object-centric debugging operations that deal with object interactions.

Halt on call. When any of an object's methods is called from any other object, execution should be halted. This operation can be applied to one or several objects and can be scoped to apply to a single method or to several ones.

Halt on invoke. When an object invokes any method, execution should be halted. This operation can be applied to one or several objects and can be scoped to apply to one or several method declarations.

Halt on creation. Execution is halted when an instance of a certain class is created.

Halt on object in invoke. Execution is halted when an object's method is invoked and a particular object is present in the invocation parameters. This operation can be applied to all methods that can be invoked on an object or on a subset of them.

Halt on object in call. When a particular object is used as a parameter of a method call the execution should be halted. This operation can be applied to all called methods or to a subset of them.

Halt on interaction. Every time two particular objects interact by one invoking a method of the other the execution is halted.

8.3 Examples: addressing debugging challenges

In this section we demonstrate how object-centric debugging fulfills the three key debugging challenges we identified (Section 8.1): (i) intercepting object-specific state access without requiring breakpoints in call sites or on state; (ii) monitoring interactions with individual objects without requiring conditional breakpoints; and (iii) supporting lightweight, live interaction with a running system without requiring breakpoints in source code. We present three case studies and compare how stack-based debuggers are used against the advantages of using an object-centric debugger.

8.3.1 Example: Tracking object-specific side-effects

The motivating problem presented in Section 8.1 is an example of tracking the cause and location of a side effect. Pharo provides a bytecode interpreter modeled by the class `InstructionStream`. This class defines an instance variable called `pc` which models the current location of execution in the instruction stream. The class `MethodContext` is a subclass of `ContextPart`, itself a subclass of `InstructionStream`. During our development, we encountered an unexpected increase of the variable `pc` holds in an instance of `MethodContext`. Identifying the circumstance in which a side-effect occurs is known to be difficult [Banning, 1979; Dolado *et al.*, 2003]. Debuggers are often employed to understand the cause of execution effect [Sillito *et al.*, 2006].

With a conventional debugger, it takes 18 *step in* operations to reach the first modification of the `pc` instance variable, and over 30 operations to reach the next one. Setting breakpoints in all possible call sites that might access `pc` does not offer any improvement: 31 of the 38 methods defined on `InstructionStream` access the variable, comprising 12 assignments; the instance variable is written 9 times in `InstructionStream`'s subclasses. In addition, the instance variable `pc` has an accessor that is referenced by 5 intensively used classes.

Object-centric debugging solves this problem trivially: by applying the *halt on write* debugging operation on the `MethodContext` instance, the source of the problem is quickly identified. Since this operation can be scoped to a specific instance variable, we can specify that execution should halt only on a *write* of the `pc` instance variable.

We can observe in Figure 8.1 how object-centric debugging differs from conventional debugging. In the upper part of Figure 8.1 we observe a traditional stack-centric debugger which is manipulated using step-wise operations. In the lower part of Figure 8.1 we observe two different object-centric debugging scopes for the same example. In one case we apply the *halt on call* and *halt on write* to the `InstructionStream` class object, thus we get the debugger to take into account messages to the class that perform these operations. In the other case we apply the same two operations to an instance of the class `InstructionStream`. The debugger takes into

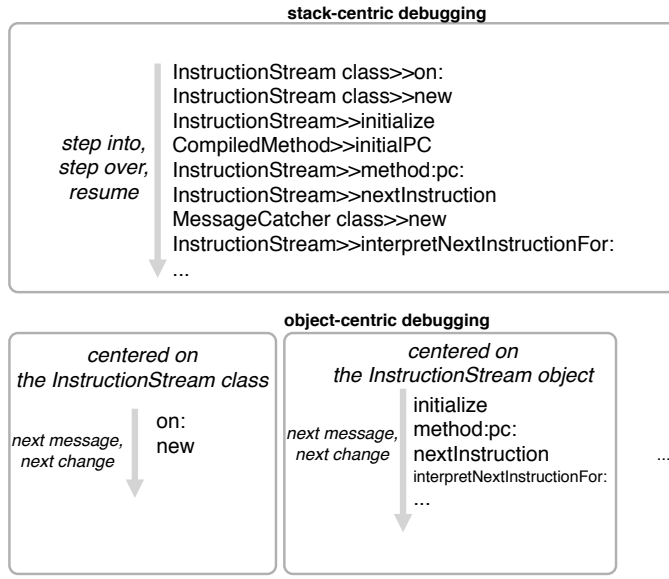

Figure 8.1: Evolution from stack-centric to object-centric debugging.

account the method calls and the instance variables changes happening in this particular object. The results of object-centric debugging are more concise and directly related to the developer's needs. With object-centric debugging we flow through the execution and see only the points that are relevant to us. With stack-centric debugging we see the whole execution and we need to steer the execution by manually introducing breakpoints.

This case study illustrates how object-centric debugging *intercepts object-specific state access without needing breakpoints to be set at call sites or on state*.

8.3.2 Example: Individual Object Interaction

Modifications to compilers can introduce subtle bugs that are very hard to understand and track down. The compilation process of Pharo Smalltalk transforms source code to bytecode. In a first phase the source code is parsed and transformed to an Abstract Syntax Tree (AST) which, afterwards, is processed by the bytecode generator. It can be cumbersome and extremely complicated to debug during the compilation process. ASTs are traversed using the visitor pattern [Gamma *et al.*, 1995]. The tree is analyzed several times for different purposes, like semantic analysis, closure analysis and early optimizations. This data is used by the `BytecodeGenerator` to produce the bytecode representation. At bytecode level variables are accessed through indices. In a compiled method, variables might have different indices depending

on the context in which they are being used. For example, a variable can have index 3 in the outer scope of the method, but index 2 in an inner scope.

Instances of `LexicalScope` model the different scopes in a particular method, mapping each variable to its index in that scope. A common bug we have encountered when modifying the compiler is to produce variables with the wrong accessing index in the bytecode, thus leading to unpredictable behavior. To debug this situation, we need to be able to track a single AST node, intercept all the messages it receives. This should enable us to see why the `LexicalScope` instances indexed are incorrect.

Analyzing the visitor patterns in a stack-based debugger is sometimes difficult due to the number invoked methods back and forth between the objects and the visitor. Moreover, we are interested in analyzing the indexing of a single variable. To be able to follow a single AST node we need to place breakpoints in all potential methods in which the node might be called, including inherited methods. There are up to 523 methods that can be invoked on instances of the class `ASTVariableNode`, rendering this approach impractical. Moreover, in a class-based system like Pharo Smalltalk, placing a breakpoint in a particular method affects all instances of `ASTVariableNode`. Conditional breakpoints could be used, however, we need to manually deal with the identity of the object and still introduce them in all the methods that may be possibly invoked (523 methods).

Object-centric debugging offers a high-level operation called *halt on call*. This operation allows method calls on a particular object to be intercepted. Using this operation we are able to follow a particular instance of `ASTVariableNode` and detect why a `LexicalScope` in the compilation process was producing an erroneous index. We can obtain the problematic instance of `ASTVariableNode` by inspecting the AST tree. In this case, the method `name` was being used by a particular visitor in charge of the indexing. The indices in an instance of `LexicalScope` were wrongly calculated due to a string assignment error in the name of the variable.

With this case study we show how object-centric debugging can *monitor interactions with individual objects without requiring conditional breakpoints*.

8.3.3 Example: Live Object Interaction

Mondrian [Meyer *et al.*, 2006] is an open and agile visualization engine. Mondrian models visualizations as graphs, *i.e.*, in terms of nodes and edges modeled by classes `MONode` and `MOEdge`. Generally, Mondrian visualizations are composed of hundreds to thousands of nodes and edges. The rendering involves a complex interaction between the various entities. When a particular node is not being rendered correctly, it can be very difficult to debug.

The rendering of Mondrian entities is performed by a `Shape` object. Each node passes itself as a parameter to a `Shape` object that specifies the rendering (double dispatch). In the case of an abnormal rendering for a particular node, traditional debuggers

promote the insertion of a breakpoint in the rendering method. However, the execution will be halted each time Mondrian renders a node. This is clearly impractical for large graphs.

Conditional breakpoints might help in this situation. To achieve this the object being tracked somehow has to be globally accessible. In languages like Java, C, C# and Smalltalk, conditional breakpoints have to be defined separately and we cannot build conditions depending on a manually selected dynamic value.

The debugger has no operations to insert at runtime conditional breakpoints that are object specific. This means that objects that are not active in the current state can only be accessed with the help of globals.

Object-centric debugging offers a high-level operation called *halt on object in call*. We apply this operation to the `Shape` object performing the rendering and we specify the `MONode` instance that we want to analyze. We obtain the `Shape` object by invoking a method on the Mondrian easel which models the plane in which nodes and edges are rendered. We can inspect a Mondrian graph visualization by clicking on each node and obtaining the object it represents. In this case the abnormally-rendered node is not being rendered with the correct size. We select that object from the visualization and thus obtain the `MONode` instance. We assume that any object constructed at runtime can be reflectively accessed and used by object-centric debugging operations. Every time that the node is passed as parameter of a method call by the particular `Shape` object, execution will be interrupted. No conditional breakpoints have to be manually defined. We also avoid dealing with object identity, and we avoid relying on the static representation of the objects.

With this case study we show how object-centric debugging can *support lightweight, live interaction with a running system without requiring breakpoints in source code*.

8.4 Implementation

There were two main implementation requirements for object-centric debugging. First, the execution of high-level debugging operations should not break other development tools such as code browsers and versioning tools. Second, we need to instrument the application to insert object-specific breakpoints at locations of interest. Because of this, we need to rely on the meta-programming facilities of the host language. These facilities are not always uniform and require ad hoc code to hook in behavior. To avoid this drawback we decided to use a framework that provides uniform meta-programming abstractions.

The prototype of object-centric debugging is built on top of the Bifröst reflection framework. From an implementation point of view, object-centric debugging requires a mechanism for runtime method redefinition. Object-specific behavior can be built on top of this mechanism.

8.4.1 Debugging Operation Definition

Each debugging operation is defined as a method in the `Object` class. Due to this, these operations can be executed on any object of the system.

In the next snippet of code we can observe the *halt on call* operation definition.

```
1 haltOnCall
2    | aMetaObject |
3    aMetaObject := BehavioralMetaObject new.
4    aMetaObject
5       when: ( MessageReceiveEvent new )
6       do: [ self metaObject unbindFrom: self.
7            TransparentBreakpoint signal ].
8    aMetaObject bindTo: self
```

Listing 8.1: Pharo Smalltalk implementation of *Halt on call* object-centric operation.

In line 3 a behavioral meta-object is instantiated. Behavioral meta-objects work by perceiving the execution of the system as a set of events like: message send, received message, state read, state write, object creation, *etc.* We use this meta-object to instrument a particular object behavior when it receives a message. The message `when:do:` defines that when a particular event happens to an object then we want a particular behavior to be executed. The class `MessageReceiveEvent` models the event when an object receives a message. The second argument is a block with the instrumentation behavior. This instrumentation is divided in two steps. First, in line 6 the instrumentation is removed from the object by unbinding it from the meta-object. Second, in line 7, a `TransparentBreakpoint`, an exception used as a breakpoint by the Smalltalk environment, is signaled thus triggering the debugger. In line 8 the meta-object that defines the adaptation is bound to the object that received the message `haltOnNextMessage`. The instrumentation behavior in lines 4–7 will only be executed when the object bound to the meta-object receives a message. In this case since we are not defining any particular message name; the instrumentation will be executed when any message is received by the adapted object. To instrument an object for a particular message name, the message `when: anEvent in: aMessageName do: aBlock` should be used instead. There is already an object-centric debugging operation defined in `Object` which does exactly that: `haltOnCall: aMessage subjectTo: aBlock`.

In the next snippet of code we can observe the *halt on write* operation definition.

```
1 haltOnWrite
2    | aMetaObject |
3    aMetaObject := BehavioralMetaObject new.
4    aMetaObject
5       when: ( StateWriteEvent new )
6       do: [ self  metaObject unbindFrom: self.
7            TransparentBreakpoint signal ].
```

```
8    aMetaObject bindTo: self
```

Listing 8.2: Pharo Smalltalk implementation of *Halt on write* object-centric operation

As we can see the definition is almost identical to Listing 8.1 but with a different meta-event. The class `StateWriteEvent` models the event when an object's instance variable is changed. This particular example instruments an object to trigger a halt when any instance variable is changed. For specifying a particular instance variable the object-centric operation `haltOnWriteFor: aVariableName` of the class `Object` should be used instead.

8.4.2 Extending Operations

Bifröst meta-objects provide facilities to manage the extension to which the adaptation should be applied. When a particular event is triggered the instrumentation block can reify various abstractions which will only be known at runtime.

```
1 aMetaObject
2       when: ( MessageReceiveEvent new )
3       do: [:receiver :selector :arguments | ... ].
```

In line 3 we can observe that the receiver, selector and arguments of the message received will be available as arguments of the block. The developer can use these arguments for evaluating conditions at runtime and define new and more specific object-centric debugging operations.

8.4.3 User Interface Modifications

To facilitate the use of object-centric debugging features the Pharo debugger and inspector were modified. The debugger was enhanced with direct buttons for *halt on call* and *halt on write*. We added menu items to the inspector with direct access to the object-centric operations. From the debugger, a developer may thus inspect any object in the current context, and from the inspector apply object-centric operations to objects of interest.

A key requirement of our implementation is not to break the existing toolchain. Smalltalk is a class-based language, so code browsers show the method definitions for each class. Object-specific modifications of the code are not well-suited to these browsers, so object-centric debugging operations are only available in the debugger and inspector.

8.5 Feasibility of Object-centric Debugging in other languages

Dynamically modifying the behavior of individual objects is an essential ingredient for implementing an object-centric debugger. This section revises the available approaches for that purpose outside Pharo Smalltalk.

Iguana [Gowing and Cahill, 1996] offers selective reification making it possible to select program elements down to individual expressions. It also allows dynamic changes to be applied in an object-specific manner. Iguana is developed for C++ and works by placing annotations in the source code to define behavioral reflective actions.

Java is a class-based object-oriented language with good support for introspection but poor support for intercession. However, several tools and techniques have been developed to overcome this limitation.

Iguana/J [Redmond and Cahill, 2002; Redmond and Cahill, 2000] is the implementation of Iguana for Java. Iguana/J enables unanticipated changes to Java applications at run-time without requiring instrumentation or restarting the application for the changes to be available. Object-specific adaptation behavior is built into the VM modifications provided by this tool.

Partial Behavioral Reflection was introduced by Tanter *et al.* [Tanter *et al.*, 2003]. This model is implemented in Reflex for the Java environment. The key advantage is that it provides a means to selectively trigger reflection, only when specific, predefined events of interest occur. Object-specific behavior can be introduced at runtime with conditional instructions in the adapted behavior.

Developers can define object-centric debugging operations and offer them through the Java Debugging Interface (JDI). It is then up to the IDE, *i.e.*, Eclipse, IntelliJ IDEA or NetBeans, to provide a user interface for object-centric actions in the debuggers.

Aspect-Oriented Programming (AOP) [Kiczales *et al.*, 1997b] modularizes cross-cutting concerns. Join points define all locations in a program that can possibly trigger the execution of additional cross-cutting code (advice). Pointcuts define at run-time whether an advice is executed. AOP features have been introduced in various languages thus making object-centric debugging feasible in these languages. Recently, new advances in AOP, like AspectWerkz [Bonér, 2004] and EAOP [Douence *et al.*, 2001], provide dynamic aspects that can be defined at runtime for specific objects. Object-centric operations can be then modeled by advice containing a breakpoint.

Self [Ungar and Smith, 1987] is a prototype-based language which follows the concepts introduced by Lieberman [Lieberman, 1986]. In Self there is no notion of class; each object conceptually defines its own format, methods, and inheritance relations.

Objects are derived from other objects by cloning and modification. Modifications can be applied to a specific object at runtime.

Ruby [Matsumoto, 2001] introduced mixins as a building block of reusability, called modules. Modules can be applied to specific objects without modifying other instances of the class adding or modifying state and methods. Object-centric operations can be modeled as modules for modifying the behavior of a particular method of an object introducing breakpoints.

Object-centric debugging can be achieved by using other techniques than a purely reflective solution as the one built on top of Bifröst.

8.6 Related Work

In recent years researchers have worked on enhancing debuggers to address the questions the developers ask themselves. In this section we review research related to object-centric debugging.

Breakpoint generation Most development environments offer convenient breakpoint facilities, however the use of these environments usually requires considerable effort to set useful breakpoints. Determining the location to insert a breakpoint entails programmer knowledge and expertise. Breakpoint generation has been proposed to reduce the effort required to select the location to insert breakpoints [Zhang *et al.*, 2010] by identifying the execution path commonly taken by failed tests. This approach uses dynamic fault localization techniques to identify suspicious program statements and states, through which both conditional and unconditional breakpoints are generated.

Dynamic languages The popularity of dynamic web content produced a number of debugging techniques for dynamic languages and web pages. *Web page breakpoints* [Barton and Odvarko, 2010] are conditional breakpoints dedicated to the web domain. For example, this approach proposes operations like "Break on attribute change" and "Break on element removal". The authors added domain-specific breakpoint capabilities to a general-purpose debugger for Javascript allowing the developer to initiate the debugging process via web page abstractions rather than lower level source code views.

Omniscient debugging Omniscient debugging [Lieberman, 1987; Lewis, 2003; Hofer, 2006] is also known as back-in-time debugging or reversible debugging. These debuggers record the whole history, or execution trace, of a debugged program. Developers can explore the history by simulating step-by-step execution both forward and backward. However, omniscient debugging has scalability issues due to the

large number of traces to manage and the challenge of quickly responding to queries on these. To overcome these issues Pothier *et al.* [Pothier *et al.*, 2007] proposed a *trace oriented debugger* (TOD) in the context of Java. TOD is composed of an efficient instrumentation for event generation, a specialized database for scalable storage, and support for partial traces to reduce trace volume. While this approach has the benefit that no data is lost, its drawback is that it requires extensive hardware power, which is not available for many developers today.

Lienhard *et al.* [Lienhard *et al.*, 2008] presented a practical approach to back-in-time debugging using partial traces in a different way than TOD. Information about objects that are eligible for garbage collection is discarded. Performance is also significantly better than in TOD because this approach is implemented at the virtual machine level, whereas all previously mentioned approaches are based on bytecode instrumentation. This approach stores historical data directly in the application memory, so does not require any additional logging facility to gather and store data.

In *query-based debugging* the user defines a query in a higher-level language that is then applied to the logged data [Martin *et al.*, 2005; Lencevicius *et al.*, 1997; Potanin *et al.*, 2004; Ducasse *et al.*, 2006a]. Queries can test complex object interrelationships and sequences of related events.

Some back-in-time debuggers instead of saving the execution data replay the program until a desired point in the past. The main advantage of replay-based approaches over logging-based approaches is their low performance overhead. Debuggers like Bdb [Feldman and Brown, 1988] and Igor [Boothe, 2000] take periodic state snapshots to optimize the time required to reach a particular point in the past. A drawback of replay-based approaches is that deterministic replay cannot be guaranteed depending on the behavior of program.

Omniscient debugging looks backwards to analyze the static history of a debugged program. Object-centric debugging looks forward to analyze the relationships between objects. Object-centric debugging avoids these scalability issues by using a runtime object-specific operations. Object-centric debugging can answer the same questions as Omniscient debugging without the scalability issues.

8.7 Conclusion

In this chapter we have presented a new debugging approach called object-centric debugging. By focusing on objects, natural debugging operations are defined to answer developer questions related to runtime behavior. Object-centric operations directly act on objects by intercepting access to runtime state, monitoring how objects interact, and supporting live interaction. Object-centric debuggers avoid the object paradox.

We demonstrated that the results of object-centric debugging are more concise and directly related to the developer's needs. With object-centric debugging we flow

through the execution and see only the points that are relevant to us. In contrast, with traditional stack-centric debuggers we see the whole execution and we need to steer the execution by manually introducing breakpoints.

We have presented a fully working prototype of an object-centric debugger and shown how this debugger is used to solve three non-trivial realistic examples. The Smalltalk prototype implementation has shown the feasibility of this approach. The impact on performance due to instrumentation is not perceived by the user. Since the history of the execution is not saved both performance and memory consumption are not as important as in omniscient debugging approach.

We have discussed how other mainstream languages can provide object-centric debugging thus demonstrating that this approach is not limited to a single language.

Chapter 9

Reflect As You Go

In this chapter we concentrate on the scoped reflection requirement. We show how object-centric reflection helps scoped reflection mechanims to avoid the object paradox.

Software systems must typically be adapted to enable software analyses such as coverage, performance, or feature analysis. It may not be possible to predict in advance which parts of the system need to be adapted, in which case either too much is adapted, or one risks to miss important parts of the system under analysis. Adaptation can therefore be both costly and awkward. We propose to avoid these problems by adapting systems on the fly. Only the entry points of the application are initially adapted with the help of reflective meta-objects that intercede on behalf of the adapted object. Each adaptation triggers further adaptations of objects reached during a run. We support adaptive software analyses by reifying the dynamic scope itself and execution events of running applications. Long-lived analyses are supported by decoupling deactivation and deinstallation of adaptations from the dynamic scope of an individual run. Multiple adaptations can be supported in a single running system, since the meta-objects keep track of the scope of each adaptation. As a consequence, only the code that needs to be adapted is touched, and the various adaptations exist in different dimensions. We present *Prisma*, an implementation of on-the-fly reflective software adaptation, we present examples of analyses supported by Prisma, and we demonstrate that Prisma is cost-effective from a performance perspective.

This chapter is structured as follows: in Section 9.1 we analyze the challenges for dynamic adaptation in the context of long-lived software analyses. We introduce the Prisma approach in Section 9.2. In Section 9.3 we take a closer look at *live feature analysis* and *back-in-time debugging*, two motivating use cases for dynamic adaptation. We present the design and implementation of Prisma in Section 9.4 and in Section 9.5 present performance benchmarks. We discuss related work in Section 9.6, and in Section 9.7 summarize the results and conclude with some remarks on future work.

9.1 Challenges for Dynamic Adaptation

Dynamic adaptation of a running software application entails the propagation of an adaptation to code that is reached dynamically by an executing thread. A propagation condition determines whether the adaptation is propagated further or not. Dynamic adaptations are particularly interesting to perform various kinds of software analysis, such as profiling, or coverage analysis. *Long-lived analyses* can especially benefit from dynamic adaptation, since the cost of installing adaptations and rerunning the analysis can be prohibitive.

In this section we consider one kind of long-lived analysis, namely *live feature analysis*, to elicit three general challenges for dynamic adaptation. The design of object-oriented applications typically reflects domain concepts, but not the features seen by end users. As a consequence, developers may be at pains to determine which software components support which features. Feature analysis attempts to recover this information, typically by instrumenting the code, exercising features, and performing post mortem analysis on the resulting data. *Live feature analysis* [Denker *et al.*, 2010] avoids the need for post mortem analysis by gathering feature information on the running system over a longer period of time, thus reducing the amount of data to be collected, and enabling the analysis of multiple features and multiple scenarios for the same features at run time.

Dynamic adaptation can ideally support such a long-lived analysis, however there are several challenges that a suitable approach must address. In particular, we identify the need to (i) control the scope of adaptation over longer periods of time, (ii) allow multiple adaptations to coexist, and (iii) dynamically update a long-lived adaptation.

9.1.1 Controlling the scope of adaptation

Existing approaches (see Section 9.6) limit dynamic adaptations to a single dynamic extent: adaptations are deactivated and uninstalled after the execution of the dynamic extent ends.

Live feature analysis, on the other hand, can be used to gather information about features exercised over multiple runs, and multiple scenarios. Such "feature growing" can thus provide more detailed information about support for features within a system than is possible to obtain with a single run. To support feature growing, however, it is important that the adaptations that monitor features remain in place after the dynamic extent of a single run has completed. Deactivating and uninstalling an adaptation therefore needs to be logically decoupled from the dynamic scope responsible for installing it in the first place for long-lived analyses that require this. The notion that a dynamic scope should be defined by a single dynamic extent is unsuitable for long-lived analyses. Multiple dynamic extents can define a valid dynamic scope as is the case in live feature analysis.

9.1.2 Activating multiple adaptations

Typical adaptations to support dynamic software analysis are very specific to a particular task, and are normally not combined with other adaptations. Test coverage instrumentation and profiling instrumentation, for example, are not normally applied to the same system at the same time, since these may interfere in unpredictable ways. For long-lived analyses, however, it would be attractive, perhaps even essential, to be able to combine analyses. Live feature analysis, for example, loses its benefits if it must be deactivated to perform test coverage analysis or profiling. Furthermore, to properly support feature growing, it should be possible to gather information about multiple features over time. The coexistence of multiple adaptations over the same objects requires a scoping mechanism to avoid interference between analyses.

9.1.3 Dynamically updating adaptations

The degree, nature and scope of information gathered by an adaptation may need to change over time.

The conditions under which adaptations are propagated may need to be refined, for example, with live feature analysis, one might initially focus only on components responsible for business logic, and later wish to assess core libraries as well. Rather than scrapping all the information gathered and restarting, one may simply adapt the propagation condition.

Alternatively, the adaptation itself may need to be changed to gather more information, for example, which specific instances of components are involved in particular features, rather than just the classes of components. A traditional problem in feature analysis is the amount of data gathered by the analysis. The adaptation can be updated to avoid saving data that is not essential to the analysis objectives thus reducing the memory footprint.

In long-lived analyses, restarting and readapting a use case can be costly. A mechanism to dynamically update adaptations is therefore key to supporting such analyses.

9.2 Prisma in a Nutshell

Prisma is an approach to dynamically adapt running software systems to support various forms of dynamic analysis. Prisma uses *reflective meta-objects* to adapt the behavior of objects at run time. *Execution is modeled as a sequence of events* which trigger the reflective meta-objects. Prisma explicitly *reifies execution runs* to manage the adaptation process. Dedicated *propagation meta-objects* assume responsibility for propagating adaptations to objects accessed within a given run. Adaptations are

scoped to a particular run, so multiple adaptations can be installed without risk of interference. *Installation and deinstallation are decoupled*, so adaptations can be retained for long-lived analyses. Adaptations are *thread-local*, but can optionally be made global.

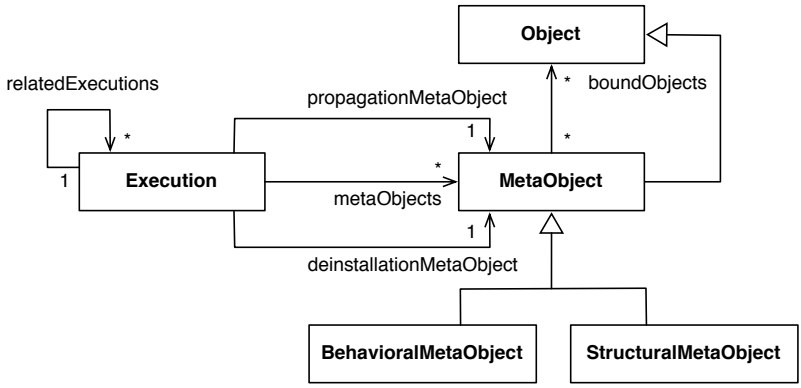

Figure 9.1: Prisma's object model.

Reflective meta-objects. Prisma makes use of reflective meta-objects known from the domain of reflective architectures [Maes, 1987b] to adapt software systems on-the-fly. Such meta-objects control various aspects of reflection offered by the underlying programming language. Meta-objects provide the implementation of adaptive behavior which is invoked at specific locations in the base system.

These reflective meta-objects address the first challenge: *dynamic adaptive adaptations*. The adaptations defined in the meta-object can change at any point during the life time of the target system. As the dynamic scopes propagates through the system the a meta-object can be changed thus changing the adaptation of the next object reached by the dynamic extent. When previously adapted objects are reached since the a meta-object was changed, the adaptation is applied anew on the original behavior, not on the adapted one.

Prisma views a running system from the perspective of operational decomposition [McAffer, 1995b], which means that execution is modeled as a sequence of meta-events, such as *invoke method* and *access state*. Meta-objects are triggered when a particular meta-event occurs. Behavioral meta-objects adapt the behavior of an object, whereas structural meta-objects adapt an object's structure, for example, to add or remove instance variables or methods.

Execution Reification. In many programming languages it is possible to reify abstractions such as activation records, execution contexts, and even the execution stack, but the concept of an *execution run* remains implicit. Prisma models execution

runs explicitly to scope adaptations to a specific set of objects reachable from a particular starting point. An execution run represents a live scope in which adaptive reflective changes take place.

An *execution* (see Figure 9.1) is composed of a set of *meta-objects*, each of which adapts a number of *bound objects*. Since a meta-object is an object, it can also be adapted by meta-objects. Meta-objects can be structural or behavioral. An execution models a dynamic scope whose starting point is an expression defining a dynamic extent.

Reified execution scopes address the first point of the *scope control for long-lived adaptations* challenge. An explicit execution scope allows two developers to use the same scope starting from two different dynamic extents. Two developers exercising the same feature can be scoped in the same domain using the same adaptations even though they are exercising the features from different parts of the system. Moreover, *combining multiple adaptations* is controlled in a finer way since explicit executions can be useed to control the interactions of different analyses. Executions prevent analyses from seeing each others' adaptions but at the same time executions can be dynamically changed to be able to allow adaptations to see their adaptations.

Propagation. A dedicated *propagation meta-object* is responsible for propagating adaptations to the dynamic extent of an execution run. When an execution run is started the first object, *i.e.*, the one receiving the first message, is adapted with the meta-objects composing the execution. One of these meta-objects is the propagation meta-object (Figure 9.1), which adapts an object so that every method call to another object causes the execution's meta-objects to be applied to that other object. An *activation condition* can be provided to restrict which objects the adaptation should be applied to.

Execution Scoping. When an object is adapted within a particular execution run this adaptation only affects other objects in the same run. When a meta-object adapts a method of an object under a specific execution run the method is copied and the adaptation is applied to that copy. As a consequence, there can be multiple versions of the same method for a given object depending on the number of scoped executions. The meta-object is responsible for managing the different method versions. When the adapted method is invoked under a particular execution run, (i) the invoked object delegates the execution of the method to the meta-object, (ii) the meta-object obtains the identity of the current execution, and (iii) with that identity it selects the version of the method to be executed. If there is no enclosing adaptive execution then the normal method lookup is used. This mechanism addresses the third challenge *combining multiple adaptations*. We can dynamically introduce multiple adaptive analyses which will not interfere with each other if applied on different execution runs scopes. However, if they are applied in the same execution run the analysis will be able to see the other adaptations.

Explicit deinstallation. By default adaptation continues until the dynamic extent ends, at which point all adaptations are uninstalled with the help of the *deinstallation meta-object*. Alternatively, an explicit *deactivation condition* can be specified to indicate when adaptations should be uninstalled. As a consequence, adaptations that have not been uninstalled can be "reused" for future analyses. This is particularly interesting for long-lived analyses that may require multiple execution runs to gather sufficient data as described by *scope control for long-lived adaptations* challenge.

Thread Locality. Thread locality determines whether an adaptation is local to a single thread or global to all running threads. Since a single execution run may make use of multiple threads, thread locality may or may not be appropriate for a given analysis. The propagation meta-object is responsible for propagting adaptations to newly-created threads, if this is desired.

9.3 Case Studies

In this section we will consider two case studies of dynamic analyses which illustrate how the challenges are addressed by the Prisma approach to dynamic, scoped adaptation.

The first is *live feature analysis*, in which software artifacts that implement a given feature are identified by instrumenting the system and exercising those features. Prisma avoids the need to statically instrument the entire system. Furthermore, multiple features can be exercised at the same time, since Prisma scopes the effect of adaptations to individual execution runs avoiding undesired adaptation interactions. The data gathered by the feature analysis can be dynamically changed to support dynamic updating of adaptations. Finally, Prisma naturally supports the ability to "grow" the feature analysis over multiple runs, since adaptations may be retained.

The second case study is *back-in-time debugging*, a technique that allows developers to step both forward and backward through an entire execution run. We show how the *object-flow analysis* approach to back-in-time debugging, previously supported by VM modifications, is easily implemented using Prisma. Controlling the scope to which particular objects should be adapted helps the analysis to reduce the size of history information. Moreover, the Prisma back-in-time debugging implementation does not interfere with the rest of the application, only with the objects reached by the dynamic extent. Adaptations can also be reused to avoid multiple adaptations of the same objects. Section 9.5 analyze the performance impact of our approach.

9.3.1 Live Feature Analysis

A feature represents a functional requirement fulfilled by a system from the perspective of a user. Since many maintenance tasks are expressed in terms of features, it is important to establish the correspondence between a feature and its implementation in source code. Traditional approaches to establish this correspondence exercise features to generate a trace of run-time events, which is then processed by post-mortem analysis. These approaches typically generate large amounts of data to analyze. Due to their static nature, these approaches do not support incremental or interactive analysis of features.

Live Feature Analysis proposes a radically different approach that provides a model at run time of features [Denker *et al.*, 2010]. This approach analyzes features on a running system and also makes it possible to "grow" feature representations by exercising different scenarios of the same feature, and identifies execution elements even to the sub-method level.

In contrast to typical dynamic feature analysis approaches, live feature analysis does not need to retain a large trace of executed data. This is because the analysis is live rather than post-mortem. Live feature analysis focuses on exploiting feature knowledge directly while the system is running. Instead of recording traces, the analysis tags with a feature annotation all the AST nodes that are executed as a result of invoking features at run time. This analysis can annotate every statement that participates in the behavior of a feature. To achieve this, we define a meta-object which specifies that when the associated AST node is executed the `FeatureTagger` object should be called to annotate the AST node.

The adaptation is achieved without any anticipation and at run-time. The user, however, still needs to specify where this adaptation should take place before exercising the features. This is a key drawback of live feature analysis as it was originally proposed [Denker *et al.*, 2010].

Prisma aids the user when the target of the feature analysis is unknown. We need to define the same meta-object used by live feature analysis inside a Prisma execution. However, the portions of the system that should be adapted are not selected by the user but by the execution.

```
loginExecution := Execution new.
loginExecution when: ASTNodeExecutionEvent
   do: [ :node | node addFeatureAnnotation: #login ]
```

The `Execution>>when: anEvent do: aBlock` method is responsible for adding a meta-object to the execution which should evaluate the provided block when a particular meta-event is produced. Whenever an AST node is executed for an adapted object the meta-level behavior is executed. We use the term "AST execution" figuratively, since AST nodes are not literally executed, but rather their lower level bytecode representation is. However, when we adapt an application we specify that we would

like something to happen when the bytecode that is the result of compiling a particular AST node is executed.

```
loginExecution
   executeOn: [ WebServer new loginAs: 'admin' password: 'pass' ]
```

Listing 9.1: Exercising the login feature on a web server.

Prisma applies this meta-object only to the specific method invoked during the execution. The meta-objects associated to the execution are never applied to a complete object unless the meta-object specifies so. In Listing 9.1 we are exercising the login feature on a web server. We are dynamically scoping the adaptation in the execution to the behavior in the block.

Dynamically updating adaptations

With the live feature analysis approach only one user can exercise a particular feature at any given time. Otherwise, feature collisions can be produced, since the whole application under analysis is globally instrumented.

Prisma solves this problem. Only users exercising features within the same execution run are affected by the adaptation defined for that run. While one user is analyzing the *login* feature, another could analyze the *printing* feature, without any interference occurring, even if the two features share some common components.

```
printingExecution := Execution new.
printingExecution when: ASTNodeExecutionEvent
   do: [ :node | node addFeatureAnnotation: #printing ]
```

When an activation condition is provided it should be possible to change it at run time. For example, we can define a dynamic scope only to be applied on objects instances of classes defined in the package `LiveFeatures-Model`.

```
loginExecution executeOn: aBlock
            subjectTo: [ :object |
                 object class package name = 'LiveFeatures-Model']
```

As we can see in the previous snippet, the condition accesses the object being adapted. While adapting the system at run time with this particular execution, we realize that we should also adapt objects in `LiveFeatures-Core`. To achieve this, we can change dynamically the execution activation condition as follows.

```
loginExecution updateCondition: [ :object |
               ( object class package name = 'LiveFeatures-Model' )
               or: [ object class package name = 'LiveFeatures-Core' ] ]
```

While the execution is being propagated the new condition is going to be applied. Objects already adapted will be readapted when the execution reaches them again. An object which is still defined in these two packages will remain adapted. However, if an object previously adapted is now left out of the dynamic condition then the adaptation will be removed from it.

Controlling scope of adaptation

Feature growing is one of the contributions of live feature analysis [Denker *et al.*, 2010]. Variants of the same feature can be exercised iteratively and incrementally, thus allowing the analysis representation of a feature to "grow" within the development environment. For example, we could exercise a feature multiple times with different parameters to obtain multiple paths of execution. This can be important, as the number of traces obtained can be considerable depending on the input data. For trace-based approaches this results in multiple traces being recorded. One feature is represented by multiple traces and therefore it is needed to manage a many-to-one mapping between features and traces. If the execution path differs over multiple runs, newly executed instructions will be tagged in addition to those already tagged. Thus we can use live feature analysis to iteratively build up the representation of a feature covering multiple paths of execution.

Retaining adaptations naturally supports feature growing over multiple runs. An execution adaptation defined by a dynamic scope can be reused multiple times by applying it over different dynamic extents. By retaining the adaptations after the dynamic extent has finished we are able to reuse the adaptation that is already in place, thus avoiding the need to adapt over and over again the same objects.

9.3.2 Back-in-time Debugging

Omniscient debugging [Lieberman, 1987; Lewis, 2003; Hofer, 2006] is also known as back-in-time debugging or reversible debugging. These debuggers record the whole history, or execution trace, of a debugged program. Developers can explore the history by simulating step-by-step execution both forward and backward. However, omniscient debugging has scalability issues due to the large number of traces to manage and the challenge of quickly responding to queries on these. To overcome these issues Pothier *et al.* [Pothier *et al.*, 2007] proposed a *trace oriented debugger* (TOD) in the context of Java. TOD is composed of an efficient instrumentation for event generation, a specialized database for scalable storage, and support for partial traces to reduce trace volume. While this approach has the benefit that no data is lost, its drawback is that it requires extensive hardware power, which is not available for many developers today.

Lienhard *et al.* [Lienhard *et al.*, 2008] presented a practical approach to back-in-time debugging based on *object flow analysis* (OFA) [Lienhard *et al.*, 2009], which tracks

the flow of objects through an execution run with the help of first-class *alias* objects. Information about objects that are eligible for garbage collection is discarded. Performance is also significantly better than in TOD because this approach is implemented at the virtual machine level, whereas all previously mentioned approaches are based on bytecode instrumentation. This approach stores historical data directly in the application memory, so does not require any additional logging facility to gather and store data.

Back-in-time debugging consumes a considerable amount of memory and requires extensive hardware power. OFA addresses these issues by providing a solution at VM level thus delivering better performance. However, this requires developers to use a modified VM instead of the mainstream one, which leads to compatibility issues and difficulties to carry the modifications forward to later versions of the VM.

Prisma provides a solution to these issues. By providing a back-in-time data recording adaptation, Prisma can control in which executions the adaptation should take place. Since Prisma strictly controls the scope within which these adaptations are applied, this leads to a reduction in the impact on memory and performance.

We implemented a simplified version of the OFA alias model. An alias represents an object reference. There are three types of aliases:

- `AllocationAlias` is created when an object is instantiated.
- `FieldReadAlias` is created when an object instance variable is read. It is also used for modeling the references in an argument passed to a method.
- `FieldWriteAlias` is created when an object instance variable is written.

An alias is composed of:

- *Value.* Any kind of value of the language.
- *Context.* Used to navigate to the method invocation in which the alias was created. It represents a frame on the execution stack and also is a real object on the heap.
- *Ancestor.* The ancestor of any alias is the previous alias of the value.
- *Predecessor.* The predecessor of a field write alias is the field write alias of the value previously stored in the field. Only field write aliases have a predecessor.

Figure 9.2 illustrates an example of a `Person` object with the attribute `name`. When the object is allocated at time t1, the field is initially undefined. Later, at time t2, the string "Doe" is written into the field and at t3 it is renamed to "Smith". In the OFA model, the initial undefined value is captured by an alias of type allocation and all subsequent stores into the field are captured by aliases of type *field write*. In the example the field first points to the alias of null, then to the alias of "Doe" and finally to the alias of "Smith". The key idea is that each alias keeps a reference to its

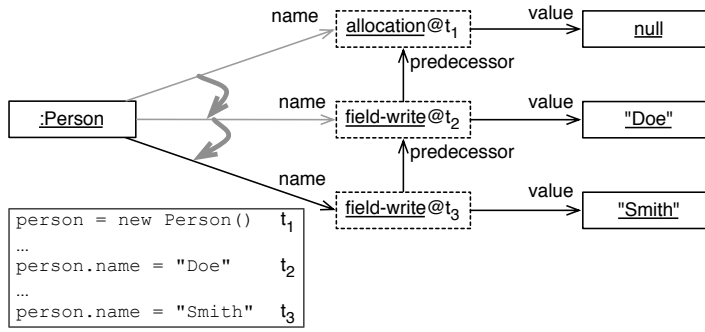

Figure 9.2: Capturing historical object state through predecessor aliases.

predecessor, that is, to the alias that was stored in the field beforehand. In this way, the alias pointed to from a field is the head of a linked list of aliases that constitute the history of that field.

The `HistoryAnalysisExecution` models the execution for the history analysis for the back-in-time debugger. This object defines the adaptation that should be applied to every object that is reached by the execution. The Prisma adaptation is divided into three parts.

- When a method is invoked on an object, a `FieldReadAlias` should be created for each method argument.
- A `FieldReadAlias` is created when a field of a particular object is read.
- When a field is written on a particular object a `FieldWriteAlias` is created.

When an object is asked for its aliases and this object has none, then an `AllocationAlias` is created on the fly for that object.

These three adaptations would normally need to be applied to the whole system or to those classes that the user thinks are important. However with Prisma the user only needs to know how to start the case study and let the execution take care of scoping the adaptation. Listing 9.2 presents the steps to create a `HistoryAnalysisExecution` and how to apply it to a particular execution run. In line 1 we create an instance of `HistoryAnalysisExecution`, in lines 2–3 we create an object and define it as the starting point of the run. The method `defaultNames` is defined as the starting selector in line 4. When the execution instance is run in line 5 the starting receiver method `defaultNames` is adapted with the reflective changes defined in the execution. These adaptations then are propagated to all objects reached during the run.

```
anExecution := HistoryAnalysisExecution new.
anObject := Person new.
anExecution startingReceiver: anObject.
anExecution startingSelector: #defaultNames.
anExecution execute.
```

Listing 9.2: History analysis creation as a Prisma execution.

```
Person>>defaultNames
   name := 'Doe'.
   self fullName: name.

HistoryObjectMock>>fullName: anObject
   fullName := anObject.
```

Listing 9.3: Code snippet being analyzed by the history analysis

For the snippet of code presented in Listing 9.3 the ancestor relationship of instance variable `name` is:

$$\begin{aligned} \text{alloc}(\text{“Doe”}) &\Longleftarrow \text{fieldWrite}(\text{“Doe”})(\text{name}) \\ &\Longleftarrow \text{fieldRead}(\text{“Doe”})(\text{name}) \\ &\Longleftarrow \text{fieldRead}(\text{“Doe”})(\text{anObject}) \\ &\Longleftarrow \text{fieldWrite}(\text{“Doe”})(\text{fullName}) \end{aligned}$$

The ancestor relationship allows the developer to flow back in time and analyze how a particular object flowed through the execution.

On the other hand the predecessor relationship allows the developer to analyze the history of values for a particular instance variable.

$$\begin{aligned} \text{alloc}(\text{“Doe”}) &\Longleftarrow \text{fieldWrite}(\text{“Doe”})(\text{name}) \\ &\Longleftarrow \text{fieldWrite}(\text{“Doe”})(\text{fullName}) \end{aligned}$$

Activating multiple adaptations

Let us consider a use case in which we have a running application and we want to apply back-in-time debugging to detect a particular bug. There are several other users working with the application and their work should not be disturbed nor should the system behavior change in any way. Due to this we cannot use the back-in-time debugger modified VM approach since the application is running too many users and the overall impact will be extremely high. Prisma can apply back-in-time debugging adaptations to a particular dynamic extent without affecting the normal application behavior and other dynamic scopes. In this scope we can select which

objects should be adapted and which information should be saved for the historical information.

At the same time other developers might need to run their own back-in-time debugger adaptations for finding specific bugs in their developed part of the application. We can choose to keep these adaptations separate in different scopes so they will not interfere with one another. Or we can reuse the same dynamic scope execution for all back-in-time debuggers.

Dynamically updating adaptations

The performance and memory footprint problem of back-in-time debuggers is well known. As we have pointed out we can selectively control which objects in the system should be adapted through the propagation. This has a positive impact on the performance. Moreover, at run time, while the scoped adaptation is in place we can selectively change the adaptation to change or reduce the data being saved. We can also change the objects that should be reached by the adaptation.

9.4 Implementation

Prisma[1] and the examples presented in this chapter are implemented in Pharo Smalltalk. Prisma is built on top of the Bifröst reflection framework.

9.4.1 Reifying Execution

A Prisma execution run models the process of executing a particular snippet of behavior. An execution is composed of Bifröst meta-objects which define how objects executed during the run should be adapted.

Beside the adaptation provided by the user there is a dedicated propagation meta-object responsible for propagating the adaptations to all objects reached during the run. This meta-object ensures that when a message send event occurs the receiver will also be adapted.

To start a scoped adaptation the user can trigger an execution on a dynamic extent:

```
anExecution executeOn: aBlock
```

1 http://scg.unibe.ch/research/bifrost/prisma/

9.4.2 Execution Scoping

Since method lookup is not reified in Smalltalk we simulate the redefinition of the method lookup. What really happens is that the method defined in the class is replaced by a set of bytecodes that delegate to the meta-object the responsibility of deciding which is the actual set of bytecodes that should be executed. If the object is not adapted then the original method in the class is executed. On the other hand, if the object has been adapted by the meta-object for the received message then the meta-object is responsible for selecting which version of the method should be executed. In Prisma the reified execution run is used as the key for deciding the method version. This technique allows the user to adapt the same object for different purposes in different execution runs avoiding adaptation conflicts.

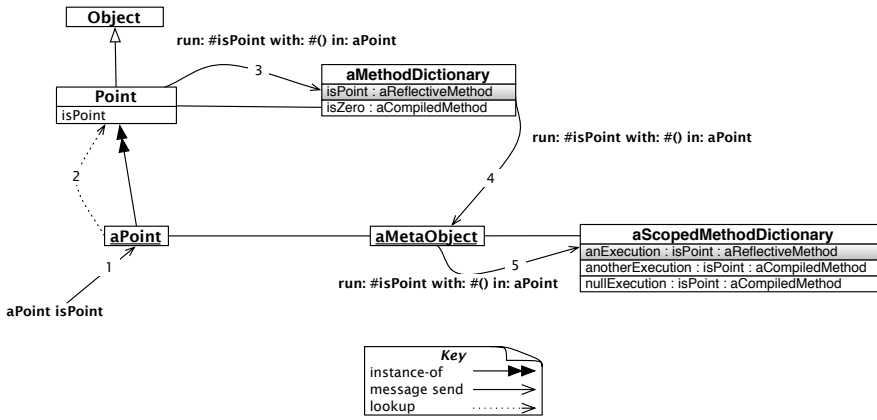

Figure 9.3: Modified method lookup for a point with a `isPoint` scoped adapted method.

In Figure 9.3 we can see an example of the modified method lookup for `aPoint>>isPoint` in a scoped environment in Prisma. First the method lookup finds the method `isPoint` defined in the `Point` class. This method is not a compiled method but a reflective method. The VM does not know how to execute this abstraction thus it delegates the execution to the reflective method itself with `run:with:in:` The responsibility of this reflective method is to delegate the execution of the method `isPoint` to the receiver's meta-object, which is done in step 4. To find the corresponding method to be executed the meta-object indexes by the execution under which the current code is running and the name of the method. The null execution models the case in which no execution is present thus this adaptation is globally visible. The meta-object finds the corresponding method which is a reflective method containing the a copy of the original AST plus adaptations. The message `run:with:in:` is sent to the reflective method which first triggers the compilation of the method, second replaces the reflective method in the method dictionary with the resulting compiled method, and finally executes the compiled method.

A dynamic variable in Smalltalk holds a value that depends on the current thread. In Pharo Smalltalk a dynamic variable can be simulated by subclassing a `Notification` which inherits from `Exception`. This dynamic variable implementation reuses the resumable exception mechanism. If the value of the dynamic variable is required at any point during the execution the exception (the dynamic variable) is signaled. The execution is triggered by `aBlock on: self do: [ :notification | notification resume: anObject ]`. The handler of the exception resumes the execution returning the object modeling the value the dynamic variable should have.

Prisma uses dynamic variables to mark the execution context in which objects are used. The execution sets the dynamic variable to be itself when it receives the message `execute`.

```
Execution>>execute
   DynamicVariable
      use: self
      during: aBlock
```

The execution value is not limited to a single thread. A single execution run can spread along several threads. This propagation is controlled by the *propagation meta-object*.

The execution abstraction should transcend the limitation of the lower level threading model. To achieve such a model the Prisma execution propagation system keeps a special adaptation for applying over spawning of new threads. This adaptation sets the execution dynamic variable in the created newly created thread to be the same in which the fork was executed.

9.4.3 Dedicated Meta-objects

Propagation meta-object. This meta-object decides which objects should be adapted while the dynamic extent unfolds. The next snippet of code shows how to change the propagation in Prisma dynamic scope:

```
anExecution propagatingWith: aMetaObject
```

By default, the propagation is defined by a meta-object that applies the adaptations on any receiver of a message send. The propagation meta-object adaptation is also applied to these objects thus producing the actual propagation. The propagation meta-object is also responsible for deciding if an adaptation should be reapplied on an already adapted object. Adaptations are not reapplied to an adapted object in the same execution scope.

Deinstallation meta-object. This meta-object is responsible for determining when the dynamic adaptation scope should finish. The next snippet of code describes how to set the deinstallation meta-object for a particular execution run:

```
anExecution deinstallingWith: aMetaObject
```

By default, scoped adaptations are not removed after the dynamic extent has finished. The user can execute `anExecution uninstall` at any point to remove the adaptations in the dynamic scope. Another option is to provide an explicit deinstallation condition. However, this approach has the disadvantage that there is no meta-level event provided by the user thus all potential triggering events points should be adapted. This has a negative impact on performance. It is simpler when the user provides a meta-object which under certain circumstances will uninstall the adaptations. Practically, the user can provide several of these meta-objects which control different features of the adapted application. This approach provides a more modular solution and avoids complex deinstallation conditions.

9.4.4 Activation Conditions

The user can define an activation condition to filter which objects the adaptation should be propagated to. The following snippet of code shows the activation condition used in feature analysis to adapt only objects whose class is defined in the packaged named `LiveFeatures-Model`:

```
anExecution executeOn: aBlock
   subjectTo: [ :object |
      object class package name = 'LiveFeatures-Model']
```

The user can update this condition dynamically as follows:

```
anExecution updateCondition: [ :object |
   ( object class package name = 'LiveFeatures-Model' )
   or: [ object class package name = 'LiveFeatures-Core' ] ]
```

Since Prisma adopts a reflective architecture it also provides another level for filtering which objects are adapted. Each meta-object added to the execution can have its own activation condition which provides a more fine-grained filtering approach. Meta-objects can therefore choose if they should be applied or not.

9.4.5 Explicit deinstallation

Adaptations are removed by unbinding meta-objects from their bound objects. This can be done at the end of the execution or immediately after the adapted object's method has been executed. The first approach is called "scarring" since it leaves

the adaptation in place to be reused in case the same execution run calls the same adapted object's method again. The second approach is called "scanning" since the adaptation moves throughout the system without leaving any mark behind.

By default, the execution uses the first approach. But the execution provides two execution methods to control this behavior: `Execution>>executeScarring` and `Execution >>executeScanning`. The adaptations are applied in the same way, and only deinstallation is affected.

9.4.6 Prisma for other languages

In this section we discuss the feasibility of implementing Prisma in other languages and environments.

Prisma's implementation can be broken down into support for the following features:

- Expressing adaptations as behavioral and structural changes at run time.
- Reflective changes can be defined unanticipatedly.
- Explicitly representing dynamic scopes to be able to reflect on them, grow and reuse.
- Avoiding scope interference.
- Supporting thread independent scopes. In Prisma the dynamic scopes are not tied to a single thread (thread local). Dynamic scopes can spread across several threads.

Having multiple scopes unfolding at the same time on a particular system requires the scoping system to keep separated various adaptations on the same objects. In a language like Java bytecode level manipulation tools like Javassist or BCEL can be used. The key point is to achieve scoped multiple versions for specific methods. Methods can be instrumented to first evaluate code implementing the decision of which method version should be executed depending on the execution run. In this way we would be simulating the method lookup redefinition. In Java it is possible to assign values to specific threads thus allowing us to associate the execution abstraction with particular threads. Extra adaptations can be introduced to detect the point in the code when a new thread is spawned and thus associating the current execution run abstraction to the newly created thread. An important problem in providing a Prisma like adaptation in the context of Java is unanticipation. Bytecode manipulation tools restrict the adaptation to take place at either compilation time or loading time thus limiting the ability of the tool to dynamically modify an object without unticipation.

As Tanter [Tanter, 2008] has pointed out, there are several techniques to support dynamic deployment of aspects: residues [Masuhara *et al.*, 2003], meta-level wrappers [Hirschfeld, 2003], optimized compilers with static analysis [Avgustinov *et al.*, 2005; Bodden *et al.*, 2007], and VM support [Bockisch *et al.*, 2004]. Moreover, there has been promising work on aspect-aware VMs [Bockisch *et al.*, 2006b; Bockisch *et al.*, 2006a] and dynamic layer (de)activation [Costanza *et al.*, 2006], suggesting that such advanced scoping mechanisms can be efficiently supported.

Recently, Moret *et al.* introduced Polymorphic Bytecode Instrumentation (PBI) [Moret *et al.*, 2011], a technique that supports dynamic dispatch amongst several, possibly independent instrumentations. These instrumentations are saved and indexed by a version identifier. A Prisma scope can be related to a particular version of instrumentations over objects' methods. When the dynamic extent is running only the method's instrumented versions indexed by the scope should be executed. However, a more static mainstream language (*i.e.*, Java) solution would likely be more static in nature. CodeMerger, the PBI implementation for Java, instruments the class library at build-time and all other classes at load-time. As such, achieving the same dynamic behavior and unanticipation as Prisma Smalltalk implementation is not possible.

9.5 Performance Analysis

In this chapter we focus on developing an approach to dynamic adaptation that is expressive enough to support practical forms of dynamic analysis, not on producing an efficient implementation for production environments. Nevertheless we have carried out some micro-benchmarks to assess the performance impact of our (unoptimized) system.

Dynamic scoping impacts the performance of the application being analyzed. We have performed a micro-benchmark to assess the maximal performance impact of Prisma. All benchmarks were performed on an Apple MacBook Pro, 2.8 GHz Intel Core i7 in Pharo 1.1.1 with the jitted Cog VM.

Let us consider the live features analysis example. We developed a benchmark where the user interaction is simulated to prevent human interaction from polluting the measurements. In this benchmark we exercised one thousand times the same feature under the same dynamic scope. This implies that the adapted objects' methods are called extensively. The results showed that the adaptation produces on average a 20% performance impact. Note that our implementation has not been aggressively optimized.

The OFA-modified VM is 7 times slower than the VM without modifications [Lienhard *et al.*, 2008]. Benchmarks suggest that a significant overhead incurs because of the additional pressure on the garbage collector. The Prisma implementation can control the scope in which the adaptations have to be installed. Furthermore, these

adaptations can be adapted, installed and uninstalled at run time. Due to this, our approach has an average of 35% slowdown. As with OFA the memory usage characteristics of each benchmark has a considerable impact on performance. However, emulating the OFA recording mechanism can produce slowdowns up to a factor of 35. This is expected since our solution does not rely on any VM optimization. Finally, when OFA recording is switched off the overhead is still 15% due to the VM modification, while in the Prisma solution there is no impact.

In conclusion, under certain circumstances Prisma OFA is faster than the VM OFA, but when the objects adapted and the data gathered grow the performance of Prisma OFA is worse. The key point is that with Prisma OFA we can control which objects should be adapted and which information should be saved. However, as we state in the chapter, fully emulating the VM OFA recording mechanism can produce slowdowns up to a factor of 35.

The AOP residues approach [Masuhara *et al.*, 2003] uses partial evaluation to find places in program text to insert aspect code and to remove unnecessary run-time checks. The same technique can be used to remove run-time checks in meta-object adaptations. In the context of Smalltalk and dynamic languages this technique has been used with good performance results in a transactional memory implementation [Renggli and Nierstrasz, 2009]. The transactional and non-transactional versions of a method coexist and inserted static checks decide which version should be used. Lienhard *et al.* [Lienhard *et al.*, 2008] have shown that modifying the VM to directly support the adaptations provides performance benefits.

These benchmarks show that Prisma approach is cost-effective from a performance perspective.

9.6 Related Work

In this section we discuss related work in the domain of controlling and adapting dynamic scope. We particularly concentrate on various systems and frameworks which provide different scoping mechanism.

Reflection, a feature common to many modern programming languages, is the ability to query and change a system at run time [Kiczales *et al.*, 1991]. Reflection is particularly useful for long-lived and highly dynamic systems since it allows them to evolve and adapt dynamically.

The original model of reflection as defined by Smith [Smith, 1982] is based on meta-level interpretation. The program is interpreted by an interpreter, which is interpreted by a meta-interpreter, and so on, leading to an unending tower of interpreters each defining the semantics of the program (the interpreter) it interprets. A tower of interpreters clearly poses performance issues in practice. To enable reflection in

mainstream languages the tower of interpreters is replaced by a reflective architecture [Maes, 1987b] where meta-objects control the various aspects of reflection offered by the language.

9.6.1 Reflective Architectures

There have been multiple approaches to define object specific adaptations. These approaches follow the per-object meta-object protocols where a meta-object modifies the behavior of the bound object [Maes, 1987b]. Composition filters are another instantiation of this same model [Bergmans and Aksit, 2001]. Composition filters can define a new program entity with behavior composed from two or more other program entities. However, there is no stack propagation proposed in these approaches. There is no way of specifying how the adaptations should me transfered to other object during execution.

ContextL is a context-oriented programming (COP) [Hirschfeld *et al.*, 2008] approach defined as an extension of CLOS. This approach provides dynamic mixin layers which are dynamically scoped. A layer is composed of structural refinements which can be dynamically activated in a dynamic extent. The definition of the scope has an explicit entry point and an implicit exit point. A mixin layer can be deactivated by using a `with-inactive-layers` expression, thus, this approach supports dynamic binding at run time. It is however impossible, to specify at deployment time a condition upon which the layer must be either uninstalled or deactivated. Activation and deactivation of layers is thread-local. A mixin layer applied in a particular thread cannot be seen by other thread unless this thread explicitly expressed that it is running under a particular layer.

Classboxes [Bergel *et al.*, 2005] and Expanders [Warth *et al.*, 2006] model class extensions that can be statically-scoped. In these approaches, a class extension is only applied to a predefined portion of the code. On the other hand, ContextL introduces the notion of dynamically-scoped class extensions. One disadvantage of this approach is that only classes and functions that are explicitly declared to be layered can partake in layered activations of new partial definitions. Thus ContextL adaptations are not unanticipated.

9.6.2 Aspect-oriented Programming

Aspect-oriented Programming (AOP) [Kiczales, 1996; Kiczales *et al.*, 1997b; Kiczales *et al.*, 1997a] is a technique which aims at increasing modularity by supporting the separation of cross-cutting concerns. Pointcuts pick out join points, *i.e.*, points in the execution of a program that trigger the execution of additional cross-cutting code called advice. Join points can be defined on the run-time model (*i.e.*, dependent on control flow). Although AOP is used to introduce changes into software systems, the focus is on cross-cutting concerns, rather than on reflecting on the system. Kiczales

et al. [Kiczales *et al.*, 1997b] claim: *"AOP is a goal, for which reflection is one powerful tool."*.

Dynamically-scoped aspects have attracted considerable attention in the object-based aspect community. Examples are their incarnation in the CaesarJ [Aracic *et al.*, 2006] language, as well as in related mechanisms outside of the pointcut-advice family, like dynamic mixin layers as provided by ContextL.

Many AOP languages provide adaptations scoped on control flow known as `cflow` pointcuts in AspectJ. The main use of this construct is to analyze the stack and when certain conditions are met regarding the stack execute and advise. `cflow` in AspectJ is by default thread local. Prisma offers a different scope mechanism since dynamic execution runs define the dynamic context. Conditions are applied not only to the control flow but also to the properties of the objects reached by the execution.

AspectS [Hirschfeld, 2003] is a dynamic aspect system defined in the context of Smalltalk. AspectS supports first-class activation conditions, which are objects modeling a dynamic condition. Since this a Smalltalk implementation the condition can be dynamically bound at runtime. This means that conditions can be installed and uninstalled at runtime. Generally, these changes are global but as Hirschfeld and Costanza [Hirschfeld and Costanza, 2006] showed thread locality can be achieved.

CaesarJ [Aracic *et al.*, 2006] provides *deploy blocks* which restrict behavioral adaptations to take place only within the dynamic extent of the block. The scope is explicitly embedded within the application code, but the this approach has an implicit exit point which is the ending of the execution of the deploy block. No value can be parameterized in the deploy block. The adaptation is bound at run time but the adaptation cannot be unbound nor rebound during the execution. Finally, the adaptation is applied locally to the thread executing the deploy block.

Following the idea of per-object meta-objects Rajan and Sullivan [Rajan and Sullivan, 2003] propose per-object aspects. An aspect deployed on a single object only sees the join points produced by this object. This join point observation can be stopped at any time.

AspectScheme [Dutchyn *et al.*, 2006] is an aspect-oriented procedural language where pointcuts and advices are first-class values. AspectScheme introduces two deployment expressions. One expression can dynamically deploy an aspect over a body expression. The other can statically deploy an aspect which only sees join points occurring lexically in its body.

Tanter [Tanter, 2008] proposes an expressive scoping model for dynamically-deployed aspects: deployment strategies. Deployment strategies provide explicit control over the propagation properties of a deployed aspect, both along call stack and delayed evaluation dimensions, as well as deployment-specific join point filters. Deployment strategies are formulated for both functional and object-oriented based aspect languages.

Tanter [Tanter, 2009] further argues that changing the point of view on scoping from a textual/dynamic dichotomy to a propagation and activation problem, enables us to formulate a general model of scoping. The author calls this new model *scoping strategies*. Tanter formalized dynamically scoped adaptations in terms of (i) the dynamic extent, (ii) the propagation function, (iii) activation conditions and (iv) the adaptations to be applied. A dynamic extent defines a dynamic scope by providing a piece of code to be executed. As the code is executed adaptations are installed, and propagated under certain conditions. The propagation function defines how the adaptations should be propagated in the dynamic extent. The use of activation conditions can further control the application of the adaptation during the dynamic scope.

Prisma follows the same idea but it addresses some of the modeling issues of this approach, like adaptation retention and dynamic scope reuse.

9.6.3 Execution Levels

The issue of infinite regression in meta-level architectures has been previously identified [des Rivières and Smith, 1984]. As a solution, Chiba *et al.* [Chiba *et al.*, 1996] present *MetaHelix*. The key point in this architecture is that levels are reified and there is a representative of each object in each level. Object extensions are layered one on top of one another. All meta-objects have a field `implemented-by` that points to a version of the code that is not reflectively changed.

Denker *et al.* [Denker *et al.*, 2008] introduced the idea of explicitly modeling the meta-level execution and the possibility to query at any point in the execution whether we are executing at the meta-level or not. An implicit *meta-context* is passed to meta-object so that they can determine at which level of execution they are.

Execution levels for aspect-oriented programming was proposed by Tanter [Tanter, 2010] to address the issue conflation in aspect-oriented programming. The author structures computation in execution levels. When fine-grained control is necessary, level shifting operators make it possible to deploy aspects at higher levels, or move computation up or down, selectively.

9.7 Conclusion

We have exemplified the limitations of previous approaches. Following the line of thought that dynamic scoping is a problem defined in terms of propagation and activation we have proposed a step further in this domain. We claimed and proved that explicit reification of the scope, in this case depicted as an execution for dynamic scopes, is key for achieving scope unanticipated deployment, scoping and condition adaptability.

We have demonstrated how Prisma is well-suited for supporting long-lived dynamic analyses. Prisma has a more fine-grained control of scope of adaptation since more than a single dynamic extent can define a dynamic scope. Multiple adaptations can coexist within the same application without interfering with each other. Finally, adaptations can be dynamically updated, avoiding the cost of deinstalling, updating and reinstalling the adaptations.

Object-centric reflection not only fulfills the scoped reflection requirement, it also allows this requirement to concentrate on objects and thus be more flexible.

Our approach is not a general solution to the scoping problem as deployment strategies and scoping strategies. Our approach is a solution for the dynamically scoped adaptation space and particularly addresses issues not solved by previous approaches. However, we plan to extend the idea of explicitly reifying the scope so as to be able to reflect and manage it dynamically and thus bring a general solution to the reflection and meta-objects domain.

We believe that research in programming languages constructs can benefit from a more flexible notion of dynamic scoping in which the scope is modeled explicitly.

Chapter 10

Conclusions

In this last chapter we summarize the contributions made by this dissertation and point to directions for future work.

10.1 Contributions of this Dissertation

Our thesis states that an object-centric reflection approach is needed to avoid the object paradox and to unify and simplify reflection.

We have presented Bifröst, an environment for engineering the meta-level through explicit meta-objects which embodies the object-centric reflection idea. Bifröst's meta-objects can be attached to any object at any time, changing its structure and behavior. Thus *partial reflection* and *unanticipated changes* are achieved. Bifröst's meta-objects are first-class objects accessible at run time. By having explicit meta-objects, *meta-level composition* can be defined for any meta-object by using composition operators.

Our contributions are the following:

1. We have demonstrated the existence of a paradox. Reflective approaches, including those with object-specific capabilities, force developers away from the runtime and the very live abstractions that they want to analyze (Chapter 1).
2. We have surveyed (Chapter 2) prior work and identified key requirements motivated by practical applications: Partial Reflection, Selective Reifications, Unanticipated Changes, Runtime Integration, Meta-Level Composition and Scoped Reflection.
3. We have presented a novel approach to meta-level engineering that organizes the meta-level into meta-objects. These meta-objects reify both structural and behavioral abstractions.
4. We have demonstrated a fully working object-centric reflection system called Bifröst and presented the implementation of non-trivial adaptations. Bifröst supports the requirements of reflection and solves the object paradox.

5. We have demonstrated how object-centric reflection addresses cannonical application of reflection: software analysis (Chapter 6, Chapter 7 and Chapter 9) and development (Chapter 5, Chapter 8 and Chapter 9).

6. We have presented talents (Chapter 5), a dynamic compositional model for reusing behavior. Talents are composed using a set of operations: composition, exclusion and aliasing.

7. We have presented Chameleon (Chapter 6), our prototype modeling the metalevel as explicit meta-events observed by development tools. Chameleon provides a dynamic model of behavioral reflection which realizes a strict separation of concerns between instrumentation and the consumers of events.

8. We have demonstrated how reflective applications like debugging, profiling and feature analysis can be redefined to be a fully dynamic (Chapter 7 and Chapter 8), closing the gap between what the developer needs at runtime and what the reflective environment provides. Due to this, the object paradox is mitigated and the user can transcend the limitations and biases of the reflective language.

9. We have presented Prisma (Chapter 9), a Bifröst extension which allows adaptation to be scoped to particular execution runs. We proved that explicit reification of the scope, in this case depicted as an execution for dynamic scopes, is key for achieving scope unanticipated deployment, scoping and condition adaptability.

10.2 Future Research Directions

Having defined an object-centric reflection approach and various tools for addressing reflection applications, we identify scope of further work in this area.

Object-Centric Reflection Applications. We plan to investigate other reflective applications; like percistency mappings, coverage, system browsing, *etc.*; to explore if object-centric reflection would have a similar impact as on debugging, feature analysis and profiling.

Scoped Talents. We plan on providing a more mature implementation of the talents scoping facilities. This technique shows great potential for the requirements of modern applications, such as dynamic adaptation and dependency injection for testing, database accesses, profiling, and so on.

Object-Centric Debugging for Statically Typed Languages. As we have demonstrated, object-centric debugging is feasible in statically typed languages, however, the lack of unanticipated changes is a key drawback. We plan to analyze

different techniques, like dynamic aspects and polymorphic bytecode instrumentation, to assess the advantages and disadvantages of this approach in statically typed languages.

General Scoped Reflection Solution. Prisma is not a general solution to the scoping problem as deployment strategies and scoping strategies. Our approach is a solution for the dynamically scoped adaptation space and particularly addresses issues unsolved by previous approaches. However, we plan to extend the idea of explicitly reifying the scope so as to be able to reflect and manage it dynamically and thus bring a general solution to the reflection and meta-objects domain.

Appendix A

Getting Started

This appendix gives instructions on how to install the Bifröst system and all its derived tools.

A.1 Bifröst Installation

There are two ways to get the Bifröst system. The recommended quick and easy way is to use the pre-built one-click distribution.

A.1.1 Downloading a One-Click Distribution

1. Download the one-click Bifröst distribution from `http://scg.unibe.ch/research/bifrost`.
2. Launch the executable of your platform:
   - Mac: `bifrost-OneClick.app`
   - Linux: `bifrost-OneClick.app/bifrost-OneClick.sh`
   - Windows: `bifrost-OneClick.app/bifrost-OneClick.exe`

A.1.2 Building a Custom Image

1. Get a Pharo-Core image from `http://www.pharo-project.org/`.
2. Execute the following Gofer script which executes the Metacello configuration of Bifröst:

```
Gofer new
   squeaksource: 'bifrost';
   package: 'ConfigurationOfBifrost';
   load.
(Smalltalk at: #ConfigurationOfBifrost)
   perform: #loadDefault.
```

A.2 Derived Tools

A.2.1 Talents

The installation and downloading instructions for talents distribution are at `http://scg.unibe.ch/research/bifrost/talents`.

A.2.2 Chameleon

The installation and downloading instructions for Chameleon distribution are at `http://scg.unibe.ch/research/bifrost/chameleon`.

A.2.3 MetaSpy

The installation and downloading instructions for MetaSpy distribution are at `http://scg.unibe.ch/research/bifrost/metaspy`.

A.2.4 Object-Centric Debugging

The installation and downloading instructions for object-centric debugging distribution are at `http://scg.unibe.ch/research/bifrost/ocd`.

A.2.5 Prisma

The installation and downloading instructions for Prisma distribution are at `http://scg.unibe.ch/research/bifrost/prisma`.

A.3 Continuous Integration Server

The latest changes and the various distributions of Bifröst and its derived tools are regularly built and tested with a Jenkins server. These Smalltalk images can be obtained at `http://scg.unibe.ch/jenkins`.

Appendix B

Bibliography

[Allan *et al.*, 2005] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 345–364, New York, NY, USA, 2005. ACM.

[Ancona *et al.*, 2000] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - A Smooth Extension of Java with Mixins. In Elisa Bertino, editor, *ECOOP 2000 Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 154–178. Springer Berlin, Heidelberg, June 2000.

[Aracic *et al.*, 2006] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135 – 173, 2006.

[Arnold and Ryder, 2001] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[Aryani *et al.*, 2011] Amir Aryani, Fabrizio Perin, Mircea Lungu, Abdun Naser Mahmood, and Oscar Nierstrasz. Can We Predict Dependencies Using Domain information? In *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE 2011)*, October 2011.

[Attardi *et al.*, 1989] Giuseppe Attardi, Cinzia Bonini, Maria Rosario Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel Programming in CLOS. In S. Cook, editor, *Proceedings ECOOP '89*, pages 243–256, Nottingham, July 1989. Cambridge University Press.

[Avgustinov *et al.*, 2005] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: An extensible AspectJ compiler. *Transactions on Aspect-Oriented Software Development*, October 2005.

[Banning, 1979] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages (POPL'79)*, pages 29–41, New York, NY, USA, 1979. ACM.

[Barton and Odvarko, 2010] John J. Barton and Jan Odvarko. Dynamic and graphical web page breakpoints. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 81–90, New York, NY, USA, 2010. ACM.

[Beck, 2002] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Longman, 2002.

[Bergel *et al.*, 2005] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Classboxes: Controlling Visibility of Class Extensions. *Journal of Computer Languages, Systems and Structures*, 31(3-4):107–126, December 2005.

[Bergel *et al.*, 2007] Alexandre Bergel, Stéphane Ducasse, Oscar Nierstrasz, and Roel Wuyts. Stateful Traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 66–90, Berlin Heidelberg, August 2007. Springer.

[Bergel *et al.*, 2008] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Creating Sophisticated Development Tools with OmniBrowser. *Journal of Computer Languages, Systems and Structures*, 34(2-3):109–129, 2008.

[Bergel *et al.*, 2011] Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-Specific Profiling. In *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 68–82, Berlin, Heidelberg, June 2011. Springer-Verlag.

[Bergmans and Aksit, 2001] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44:51–57, October 2001.

[Bergmans and Aksit, 2004] Lodewijk Bergmans and Mehmet Aksit. Principles and Design Rationale of Composition Filters. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect Oriented Software Development*, pages 63–95. Addison-Wesley, Boston, 2004.

[Bettini *et al.*, 2009] Lorenzo Bettini, Sara Capecchi, and Betti Venneri. Featherweight Java with dynamic and static overloading. *Sci. Comput. Program.*, 74:261–278, March 2009.

[Black *et al.*, 2009] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[Bobrow *et al.*, 1988] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonia E. Keene, Gregor Kiczales, and D.A. Moon. Common Lisp Object System Specification, X3J13. Technical Report 88-003, (ANSI COMMON LISP), 1988.

[Bockisch *et al.*, 2004] Christoph Bockisch, Michael Haupt, Mira Mezini, and Klaus Ostermann. Virtual Machine Support for Dynamic Join Points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[Bockisch *et al.*, 2006a] Christoph Bockisch, Matthew Arnold, Tom Dinkelaker, and Mira Mezini. Adapting virtual machine techniques for seamless aspect support. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 109–124, New York, NY, USA, 2006. ACM.

[Bockisch *et al.*, 2006b] Christoph Bockisch, Sebastian Kanthak, Michael Haupt, Matthew Arnold, and Mira Mezini. Efficient control flow quantification. In Peri L. Tarr and William R. Cook, editors, *Proceedings of OOPSLA 2006*, pages 125–138. ACM, 2006.

[Bockisch *et al.*, 2011] Christoph Bockisch, Somayeh Malakuti, Mehmet Akşit, and Shmuel Katz. Making aspects natural: events and composition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 285–300, New York, NY, USA, 2011. ACM.

[Bodden *et al.*, 2007] Eric Bodden, Laurie J. Hendren, and Ondrej Lhoták. A Staged Static Program Analysis to Improve the Performance of Runtime Monitoring. In Erik Ernst, editor, *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.

[Bonér, 2004] Jonas Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD '04, pages 5–6, New York, NY, USA, 2004. ACM.

[Boothe, 2000] Bob Boothe. Efficient algorithms for bidirectional debugging. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation (PLDI'00)*, pages 299–310, New York, NY, USA, 2000. ACM.

[Borning and Ingalls, 1982] Alan H. Borning and Daniel H.H. Ingalls. Multiple Inheritance in Smalltalk-80. In *Proceedings at the National Conference on AI*, pages 234–237, Pittsburgh, PA, 1982.

[Bouraqadi, 2004] Noury Bouraqadi. Safe Metaclass Composition Using Mixin-Based Inheritance. *Journal of Computer Languages, Systems and Structures*, 30(1-2):49–61, April 2004.

[Bracha and Cook, 1990] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, October 1990.

[Bracha and Ungar, 2004] Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

[Bracha *et al.*, 2010] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules as objects in Newspeak. In *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, ECOOP'10, pages 405–428, Berlin, Heidelberg, June 2010. Springer-Verlag.

[Bracha, 1992] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Dept. of Computer Science, University of Utah, March 1992.

[Bracha, 2004] Gilad Bracha. Generics in the Java Programming Language, July 2004. java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf.

[Brant *et al.*, 1998] John Brant, Brian Foote, Ralph Johnson, and Don Roberts. Wrappers to the Rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[Briot and Cointe, 1989] Jean-Pierre Briot and Pierre Cointe. Programming with Explicit Metaclasses in Smalltalk-80. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 419–432, October 1989.

[Bunge, 2009] Philipp Bunge. Scripting Browsers with Glamour. Master's thesis, University of Bern, April 2009.

[Cantrill *et al.*, 2004] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference*, pages 15–28, Berkeley, CA, USA, 2004. USENIX Association.

[Caromel *et al.*, 2001] Denis Caromel, Fabrice Huet, and Julien Vayssière. A simple security-aware MOP for Java. In *In Metalevel Architectures and Separation of Crosscutting Concerns, Third International Conference, REFLECTION 2001, volume LNCS 2192*, pages 118–125. Springer-Verlag, 2001.

[Cazzola, 1998] Walter Cazzola. Evaluation of Object-Oriented Reflective Models. In *In Proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS 98), in 12th European Conference on Object-Oriented Programming (ECOOP 98), Brussels, Belgium, on 20th-24th*, pages 3–540, 1998.

[Chiba *et al.*, 1996] Shigeru Chiba, Gregor Kiczales, and John Lamping. Avoiding Confusion in Metacircularity: The Meta-Helix. In Kokichi Futatsugi and Satoshi Matsuoka, editors, *Proceedings of ISOTAS '96*, volume 1049 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 1996.

[Chiba, 2000] Shigeru Chiba. Load-Time Structural Reflection in Java. In *Proceedings of ECOOP 2000*, volume 1850 of *LNCS*, pages 313–336, 2000.

[Cohen and Gil, 2009] Tal Cohen and Joseph (Yossi) Gil. Three approaches to object evolution. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 57–66, New York, NY, USA, 2009. ACM.

[Cointe, 1987] Pierre Cointe. Metaclasses are First Class: the ObjVlisp Model. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 156–167, December 1987.

[Costanza and Hirschfeld, 2005] Pascal Costanza and Robert Hirschfeld. Language Constructs for Context-oriented Programming: An Overview of ContextL. In *Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05*, pages 1–10, New York, NY, USA, October 2005. ACM.

[Costanza *et al.*, 2006] Pascal Costanza, Robert Hirschfeld, and Wolfgang De Meuter. Efficient Layer Activation for Switching Context-dependent Behavior. In *JMLC'06: Proceedings of the Joint Modular Languages Conference*, volume 4228 of *LNCS*, pages 84–103, Oxford, UK, September 2006. Springer.

[Cuadrado and Molina, 2009] Jesús Sánchez Cuadrado and Jesús García Molina. A Model-Based Approach to Families of Embedded Domain Specific Languages. *IEEE Transactions on Software Engineering*, 99(1), 2009.

[Darderes and Prieto, 2004] Betiana Darderes and Máximo Prieto. Subjective Behavior: a General Dynamic Method Dispatch. In *OOPSLA Workshop on Revival of Dynamic Languages*, October 2004.

[Demeyer *et al.*, 2003] Serge Demeyer, Stéphane Ducasse, Kim Mens, Adrian Trifu, and Rajesh Vasa. Report of the ECOOP'03 Workshop on Object-Oriented Reengineering. In *Object-Oriented Technology (ECOOP'03 Workshop Reader)*, LNCS, pages 72–85. Springer-Verlag, 2003.

[DeMichiel and Gabriel, 1987] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 151–170, Paris, France, June 1987. Springer-Verlag.

[Denker *et al.*, 2007] Marcus Denker, Stéphane Ducasse, Adrian Lienhard, and Philippe Marschall. Sub-Method Reflection. In *Journal of Object Technology, Special Issue. Proceedings of TOOLS Europe 2007*, volume 6/9, pages 231–251. ETH, October 2007.

[Denker *et al.*, 2008] Marcus Denker, Mathieu Suen, and Stéphane Ducasse. The Meta in Meta-object Architectures. In *Proceedings of TOOLS EUROPE 2008*, volume 11 of *LNBIP*, pages 218–237. Springer-Verlag, 2008.

[Denker *et al.*, 2010] Marcus Denker, Jorge Ressia, Orla Greevy, and Oscar Nierstrasz. Modeling Features at Runtime. In *Proceedings of MODELS 2010 Part II*, volume 6395 of *LNCS*, pages 138–152. Springer-Verlag, October 2010.

[Denker, 2008] Marcus Denker. *Sub-method Structural and Behavioral Reflection*. PhD thesis, University of Bern, May 2008.

[des Rivières and Smith, 1984] Jim des Rivières and Brian Cantwell Smith. The implementation of procedurally reflective languages. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 331–347, New York, NY, USA, 1984. ACM.

[Deursen *et al.*, 2000] Arie van Deursen, Paul Klint, and Joost Visser. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000.

[Di Gianantonio *et al.*, 1998] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '98, pages 166–178, New York, NY, USA, 1998. ACM.

[Dixon *et al.*, 1989] R. Dixon, T. McKee, M. Vaughan, and P. Schweizer. A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 211–214, October 1989.

[Dolado *et al.*, 2003] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An Empirical Investigation of the Influence of a Type of Side Effects on Program Comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, 2003.

[Douence and Südholt, 2002] Rémi Douence and Mario Südholt. A model and a tool for Event-based Aspect-Oriented Programming (EAOP). Technical report, Ecole des Mines de Nantes, December 2002.

[Douence *et al.*, 2001] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Heidelberg, and New York, September 2001. Springer-Verlag.

[Drossopoulou *et al.*, 2001] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Reclassification. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 130–149, London, UK, UK, 2001. Springer-Verlag.

[Ducasse *et al.*, 2006a] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-Oriented Legacy System Trace-based Logic Testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[Ducasse *et al.*, 2006b] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A Mechanism for fine-grained Reuse. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 28(2):331–388, March 2006.

[Ducournau *et al.*, 1992] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Monotonic Conflict Resolution Mechanisms for Inheritance. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27, pages 16–24, October 1992.

[Dunsmore *et al.*, 2000] Alastair Dunsmore, Marc Roper, and Murray Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[Dutchyn *et al.*, 2006] Christopher Dutchyn, David B. Tucker, and Shriram Krishnamurthi. Semantics and scoping of aspects in higher-order languages. *Sci. Comput. Program.*, 63(3):207–239, 2006.

[Eisenbarth *et al.*, 2003] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating Features in Source Code. *IEEE Computer*, 29(3):210–224, March 2003.

[Feldman and Brown, 1988] Stuart I. Feldman and Channing B. Brown. IGOR: a system for program debugging via reversible execution. In *Proceedings of the 1988 ACM SIGPLAN and SIGOPS workshop on Parallel and distributed debugging (PADD'88)*, pages 112–123, New York, NY, USA, 1988. ACM.

[Ferber, 1989] Jacques Ferber. Computational Reflection in Class-Based Object-Oriented Languages. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, volume 24, pages 317–326, October 1989.

[Flatt *et al.*, 1998] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and Mixins. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, New York, NY, USA, 1998. ACM Press.

[Fowler, 2005] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley, 75 Arlington Street Suite 300, Boston MA, 02116 USA, 2005.

[Freeman and Pryce, 2006] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in Java. In *OOPSLA'06: Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 855–865, Portland, OR, USA, 2006. ACM.

[Gamma *et al.*, 1995] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, Reading, Mass., 1995.

[Gasiunas *et al.*, 2011] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Nú nez, and Jacques Noyé. EScala: modular event-driven object interactions in scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 227–240, New York, NY, USA, 2011. ACM.

[Ghelli, 2002] Giorgio Ghelli. Foundations for Extensible Objects with Roles. *Inf. Comput.*, 175(1):50–75, 2002.

[Gîrba, 2010] Tudor Gîrba. The Moose Book, 2010.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[Goldberg and Robson, 1989] Adele Goldberg and Dave Robson. *Smalltalk-80: The Language*. Addison Wesley, 1989.

[Golm and Kleinöder, 1999] Michael Golm and Jürgen Kleinöder. Jumping to the Meta Level: Behavioral Reflection Can Be Fast and Flexible. In *Reflection '99: Proceedings of the Second International Conference on Meta-Level Architectures and Reflection*, pages 22–39, London, UK, 1999. Springer-Verlag.

[Gowing and Cahill, 1996] Brendan Gowing and Vinny Cahill. Meta-Object Protocols for C++: The Iguana Approach. Technical report, AAA, 1996.

[Gupta and Hwu, 1992] Aloke Gupta and Wen-Mei W. Hwu. Xprof: profiling the execution of X Window programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 253–254, New York, NY, USA, 1992. ACM.

[Hamou-Lhadj *et al.*, 2005] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering Behavioral Design Models from Execution Traces. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press.

[Hastings and Joyce, 1992] Reed Hastings and Bob Joyce. Purify: Fast Detection of Memory Leaks and Access Errors, 1992.

[Hirschfeld and Costanza, 2006] Robert Hirschfeld and Pascal Costanza. Extending Advice Activation in AspectS. In *Proceedings of the AOSD Workshop on Open and Dynamic Aspect Languages (ODAL)*, 2006.

[Hirschfeld *et al.*, 2008] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-Oriented Programming. *Journal of Object Technology*, 7(3), March 2008.

[Hirschfeld, 2003] Robert Hirschfeld. AspectS — Aspect-Oriented Programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, number 2591 in LNCS, pages 216–232. Springer, 2003.

[Hofer, 2006] Christoph Hofer. Implementing a Backward-In-Time Debugger. Master's thesis, University of Bern, September 2006.

[Hollingsworth *et al.*, 1997] J. K. Hollingsworth, O. Niam, B. P. Miller, Zhichen Xu, M. J. R. Goncalves, and Ling Zheng. MDL: A Language And Compiler For Dynamic Program Instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 201–, Washington, DC, USA, 1997. IEEE Computer Society.

[Ibrahim, 1991] Mamdouh H. Ibrahim. Reflection and metalevel architectures in object-oriented programming (workshop session). In *OOPSLA/ECOOP '90: Proceedings of the European conference on Object-oriented programming addendum: systems, languages, and applications*, pages 73–80, New York, NY, USA, 1991. ACM Press.

[Ingalls, 1978] Daniel H. H. Ingalls. The Smalltalk-76 programming system design and implementation. In *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 9–16, New York, NY, USA, 1978. ACM.

[Inostroza *et al.*, 2011] Milton Inostroza, Éric Tanter, and Eric Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 508–511, New York, NY, USA, 2011. ACM.

[Kabanov and Raudjärv, 2008] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *PPPJ'08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 189–197, Modena, Italy, 2008. ACM.

[Keene, 1989] Sonia E. Keene. *Object-Oriented Programming in Common-Lisp*. Addison Wesley, 75 Arlington Street Suite 300, Boston MA, 02116 USA, 1989.

[Kiczales *et al.*, 1991] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.

[Kiczales *et al.*, 1997a] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[Kiczales *et al.*, 1997b] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag.

[Kiczales *et al.*, 2001] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings ECOOP 2001*, number 2072 in LNCS, pages 327–353, London, UK, 2001. Springer Verlag.

[Kiczales, 1996] Gregor Kiczales. Aspect-Oriented Programming: A Position Paper From the Xerox PARC Aspect-Oriented Programming Project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming*. Dpunkt Verlag, 1996.

[Kothari *et al.*, 2006] Jay Kothari, Trip Denton, Spiros Mancoridis, and Ali Shokoufandeh. On Computing the Canonical Features of Software Systems. In *13th IEEE Working Conference on Reverse Engineering (WCRE 2006)*, October 2006.

[Kristensen, 1995] Bent Bruun Kristensen. Object-Oriented Modeling with Roles. In John Murphy and Brian Stone, editors, *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71, London , UK, 1995. Springer-Verlag.

[Lanza and Ducasse, 2003] Michele Lanza and Stéphane Ducasse. Polymetric Views—A Lightweight Visual Approach to Reverse Engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003.

[Lawless and Milner, 1989] Jo A. Lawless and Molly M. Milner. *Understanding Clos the Common Lisp Object System*. Digital Press, 225 Wildwood St., Woburn, MA 01801, 1989.

[Lencevicius *et al.*, 1997] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-Based Debugging of Object-Oriented Programs. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming (OOPSLA'97)*, pages 304–317, New York, NY, USA, 1997. ACM.

[Lewis, 2003] Bill Lewis. Debugging Backwards in Time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG'03)*, October 2003.

[Lieberherr *et al.*, 1999] Karl Lieberherr, David H. Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA 02115, March 1999.

[Lieberman, 1986] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 214–223, November 1986.

[Lieberman, 1987] Henry Lieberman. Reversible Object-Oriented Interpreters. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 11–19, Paris, France, June 1987. Springer-Verlag.

[Lienhard *et al.*, 2008] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical Object-Oriented Back-in-Time Debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[Lienhard *et al.*, 2009] Adrian Lienhard, Stéphane Ducasse, and Tudor Gîrba. Taking an Object-Centric View on Dynamic Information with Object Flow Analysis. *Journal of Computer Languages, Systems and Structures*, 35(1):63–79, 2009.

[Lincke *et al.*, 2011] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in ContextJS. *Sci. Comput. Program.*, 76:1194–1209, December 2011.

[Madsen *et al.*, 1993] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the Beta Programming Language*. Addison Wesley, Reading, Mass., 1993.

[Maes, 1987a] Pattie Maes. *Computational Reflection*. PhD thesis, Laboratory for Artificial Intelligence, Vrije Universiteit Brussel, Brussels Belgium, January 1987.

[Maes, 1987b] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987.

[Marschall, 2006] Philippe Marschall. Persephone: Taking Smalltalk Reflection to the sub-method Level. Master's thesis, University of Bern, December 2006.

[Martin *et al.*, 2005] Mickael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 363–385, New York, NY, USA, 2005. ACM Press.

[Maruyama and Terada, 2003] Kazutaka Maruyama and Minoru Terada. Debugging with Reverse Watchpoint. In *Proceedings of the Third International Conference on Quality Software (QSIC'03)*, page 116, Washington, DC, USA, 2003. IEEE Computer Society.

[Masuhara *et al.*, 2003] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag.

[Matsumoto, 2001] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2001.

[McAffer, 1995a] Jeff McAffer. *A Meta-level Architecture for Prototyping Object Systems*. Ph.D. thesis, University of Tokyo, September 1995.

[McAffer, 1995b] Jeff McAffer. Meta-level Programming with CodA. In W. Olthoff, editor, *Proceedings ECOOP '95*, volume 952 of *LNCS*, pages 190–214, Aarhus, Denmark, August 1995. Springer-Verlag.

[McAffer, 1996] Jeff McAffer. Engineering the Meta Level. In Gregor Kiczales, editor, *Proceedings of the 1st International Conference on Metalevel Architectures and Reflection (Reflection 96)*, San Francisco, USA, April 1996.

[Mehta and Heineman, 2002] Alok Mehta and George Heineman. Evolving legacy systems features using regression test cases and components. In *Proceedings ACM International Workshop on Principles of Software Evolution*, pages 190–193, New York NY, 2002. ACM Press.

[Mens and van Limberghen, 1996] Tom Mens and Marc van Limberghen. Encapsulation and Composition as Orthogonal Operators on Mixins: A Solution to Multiple Inheritance Problems. *Object Oriented Systems*, 3(1):1–30, 1996.

[Meyer *et al.*, 2006] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An Agile Visualization Framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.

[Meyer, 1997] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[Mezini, 1997] Mira Mezini. Dynamic Object Evolution without Name Collisions. In *Proceedings ECOOP '97*, pages 190–219. Springer-Verlag, June 1997.

[Moon, 1986] David A. Moon. Object-Oriented Programming with Flavors. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 1–8, November 1986.

[Moret *et al.*, 2011] Philippe Moret, Walter Binder, and Éric Tanter. Polymorphic bytecode instrumentation. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 129–140, New York, NY, USA, 2011. ACM.

[Nagappan, 2010] Meiyappan Nagappan. Analysis of execution log files. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 409–412, New York, NY, USA, 2010. ACM.

[Nielsen and Richards, 1989] Jakob Nielsen and John T. Richards. The Experience of Learning and Using Smalltalk. *IEEE Software*, 6(3):73–77, 1989.

[Nielson, 1989] Flemming Nielson. The Typed Lambda-Calculus with First-Class Processes. In E. Odijk and J-C. Syre, editors, *Proceedings PARLE '89, Vol II*, volume 366 of *LNCS*, pages 357–373, Eindhoven, June 1989. Springer-Verlag.

[Nierstrasz *et al.*, 2005] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The Story of Moose: an Agile Reengineering Environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York, NY, USA, September 2005. ACM Press. Invited paper.

[Oliva and Buzato, 1999] Alexandre Oliva and Luiz Eduardo Buzato. The Design and Implementation of Guarana. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 203–216, San Diego, California, USA, May 1999.

[Paepcke, 1993] Andreas Paepcke. User-Level Language Crafting. In *Object-Oriented Programming: the CLOS perspective*, pages 66–99. MIT Press, Cambridge, MA, USA, 1993.

[Perin *et al.*, 2010] Fabrizio Perin, Tudor Gîrba, and Oscar Nierstrasz. Recovery and Analysis of Transaction Scope from Scattered Information in Java Enterprise Applications. In *Proceedings of International Conference on Software Maintenance 2010*, September 2010.

[Perin, 2010] Fabrizio Perin. MooseJEE: A Moose Extension to Enable the assessment of JEAs. In *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, September 2010.

[Piumarta and Warth, 2006] Ian Piumarta and Alessandro Warth. Open Reusable Object Models. Technical report, Viewpoints Research Institute, 2006. VPRI Research Note RN-2006-003-a.

[Potanin *et al.*, 2004] Alex Potanin, James Noble, and Robert Biddle. Snapshot Query-Based Debugging. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[Pothier *et al.*, 2007] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable Omniscient Debugging. *Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07)*, 42(10):535–552, 2007.

[Rajan and Sullivan, 2003] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. *SIGSOFT Softw. Eng. Notes*, 28(5):297–306, 2003.

[Rashid and Aksit, 2006] Awais Rashid and Mehmet Aksit, editors. *Transactions on Aspect-Oriented Software Development II*, volume 4242 of *Lecture Notes in Computer Science*, Berlin, Heidelberg, 2006. Springer.

[Redmond and Cahill, 2000] Barry Redmond and Vinny Cahill. Iguana/J: Towards a Dynamic and Efficient Reflective Architecture for Java. In *Proceedings of European Conference on Object-Oriented Programming, workshop on Reflection and Meta-Level Architectures*, 2000.

[Redmond and Cahill, 2002] Barry Redmond and Vinny Cahill. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. In *Proceedings of European Conference on Object-Oriented Programming*, volume 2374, pages 205–230. Springer-Verlag, 2002.

[Renggli and Nierstrasz, 2009] Lukas Renggli and Oscar Nierstrasz. Transactional Memory in a Dynamic Language. *Journal of Computer Languages, Systems and Structures*, 35(1):21–30, April 2009.

[Renggli *et al.*, 2010a] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Domain-Specific Program Checking. In Jan Vitek, editor, *Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10)*, volume 6141 of *LNCS*, pages 213–232. Springer-Verlag, 2010.

[Renggli *et al.*, 2010b] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical Dynamic Grammars for Dynamic Languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010.

[Renggli *et al.*, 2010c] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding Languages Without Breaking Tools. In Theo D'Hondt, editor, *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag.

[Ressia *et al.*, 2010] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-Time Evolution through Explicit Meta-Objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/.

[Ressia *et al.*, 2011] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: Dynamically Composable Units of Reuse. In *Proceedings of International Workshop on Smalltalk Technologies (IWST 2011)*, pages 109–118, 2011.

[Ressia *et al.*, 2012a] Jorge Ressia, Alexandre Bergel, and Oscar Nierstrasz. Object-Centric Debugging. In *Proceeding of the 34rd international conference on Software engineering*, ICSE '12, 2012.

[Ressia *et al.*, 2012b] Jorge Ressia, Alexandre Bergel, Oscar Nierstrasz, and Lukas Renggli. Modeling Domain-Specific Profilers. *Journal of Object Technology*, 11(1):1–21, April 2012.

[Röthlisberger *et al.*, 2008] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated Partial Behavioral Reflection: Adapting Applications at Runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[Röthlisberger, 2010] David Röthlisberger. *Augmenting IDEs with Runtime Information for Software Maintenance*. PhD thesis, University of Bern, May 2010.

[Sakkinen, 1989] Markku Sakkinen. Disciplined Inheritance. In S. Cook, editor, *Proceedings ECOOP '89*, pages 39–56, Nottingham, July 1989. Cambridge University Press.

[Salah and Mancoridis, 2004] Maher Salah and Spiros Mancoridis. A Hierarchy of Dynamic Software Views: from Object-Interactions to Feature-Interacions. In *Proceedings IEEE International Conference on Software Maintenance (ICSM 2004)*, pages 72–81, Los Alamitos CA, 2004. IEEE Computer Society Press.

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Killian, and Carrie Wilpolt. An Introduction to Trellis/Owl. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 9–16, November 1986.

[Schärli *et al.*, 2003] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable Units of Behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274, Berlin Heidelberg, July 2003. Springer Verlag.

[Sillito *et al.*, 2006] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 23–34, New York, NY, USA, 2006. ACM.

[Sillito *et al.*, 2008] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and Answering Questions during a Programming Change Task. *IEEE Trans. Softw. Eng.*, 34:434–451, July 2008.

[Smith and Drossopoulou, 2005] Charles Smith and Sophia Drossopoulou. Chai: Typed Traits in Java. In *Proceedings ECOOP 2005*, 2005.

[Smith and Ungar, 1996] Randall B. Smith and Dave Ungar. A Simple and Unifying Approach to Subjective Objects. *TAPOS special issue on Subjectivity in Object-Oriented Systems*, 2(3):161–178, December 1996.

[Smith, 1982] Brian Cantwell Smith. *Reflection and Semantics in a Procedural Language*. Ph.D. thesis, MIT, Cambridge, MA, 1982.

[Smith, 1984] Brian Cantwell Smith. Reflection and Semantics in Lisp. In *Proceedings of POPL '84*, pages 23–3, 1984.

[Snyder, 1986] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 38–45, November 1986.

[Srivastava and Eustace, 2004] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. *SIGPLAN Not.*, 39:528–539, April 2004.

[Stein, 1987] Lynn Andrea Stein. Delegation Is Inheritance. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 138–146, December 1987.

[Störzer and Koppen, 2004] Maximilian Störzer and Christian Koppen. PCDiff: Attacking the Fragile Pointcut Problem, Abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, sep 2004.

[Stroustrup, 1986] Bjarne Stroustrup. *The C++ Programming Language*. Addison Wesley, Reading, Mass., 1986.

[Stuart *et al.*, 2000] Omri Traub Stuart, Stuart Schechter, and Michael D. Smith. Ephemeral Instrumentation for Lightweight Program Profiling. Technical report, Harvard University, 2000.

[Suvée *et al.*, 2003] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM.

[Sweeney and Gil, 1999] Peter F. Sweeney and Joseph (Yossi) Gil. Space and time-efficient memory layout for multiple inheritance. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 256–275, New York, NY, USA, 1999. ACM.

[Tanter *et al.*, 2003] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial Behavioral Reflection: Spatial and Temporal Selection of Reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.

[Tanter, 2006] Éric Tanter. Aspects of Composition in the Reflex AOP Kernel. In Welf Löwe and Mario Südholt, editors, *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *LNCS*, pages 98–113, Vienna, Austria, March 2006. Springer.

[Tanter, 2007] Éric Tanter. On Dynamically-Scoped Crosscutting Mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33, February 2007.

[Tanter, 2008] Éric Tanter. Expressive Scoping of Dynamically-Deployed Aspects. In *Proceedings of the 7th ACM International Conference on Aspect-Oriented Software Development (AOSD 2008)*, pages 168–179, Brussels, Belgium, April 2008. ACM Press.

[Tanter, 2009] Éric Tanter. Beyond static and dynamic scope. In *Proceedings of the 5th symposium on Dynamic languages*, DLS '09, pages 3–14, New York, NY, USA, 2009. ACM.

[Tanter, 2010] Éric Tanter. Execution Levels for Aspect-Oriented Programming. In *Proceedings of the 9th ACM International Conference on Aspect-Oriented Software Development (AOSD 2010)*, pages 37–48, Rennes and Saint Malo, France, March 2010. ACM Press. Best Paper Award.

[Tichelaar *et al.*, 2000] Sander Tichelaar, Stéphane Ducasse, Serge Demeyer, and Oscar Nierstrasz. A Meta-model for Language-Independent Refactoring. In *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*, pages 157–167, Los Alamitos, CA, November 2000. IEEE Computer Society Press.

[Tisi *et al.*, 2010] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving higher-order transformations support in ATL. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 215–229, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 227–242, December 1987.

[Visser, 2004] Eelco Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004.

[Warth *et al.*, 2006] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 37–56, New York, NY, USA, 2006. ACM Press.

[Welch and Stroud, 1999] Ian Welch and Robert Stroud. Dalang – A Reflective Java Extension, 1999. In Proceedings of the OOPSLA 99 Workshop on Reflective Programming in C++ and Java, Vancouver, Canada, Oct.

[Welch and Stroud, 2001] Ian Welch and Robert J. Stroud. Kava — Using Bytecode Rewriting to add Behavioural Reflection to Java. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technology (COOTS'2001)*, pages 119–130, San Antonio, Texas, USA, February 2001.

[Wilde and Huitt, 1992] Norman Wilde and Ross Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, December 1992.

[Wu, 1998] Zhixue Wu. Reflective Java and a reflective component-based transaction architecture, 1998. In Proceedings of the ACM OOPSLA 98 Workshop on Reflective Programming in Java and C++, Oct. 1998.

[Yaghmour and Dagenais, 2000] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association.

[Zhang *et al.*, 2010] Cheng Zhang, Dacong Yan, Jianjun Zhao, Yuting Chen, and Shengqian Yang. BPGen: an automated breakpoint generator for debugging. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 271–274, New York, NY, USA, 2010. ACM.